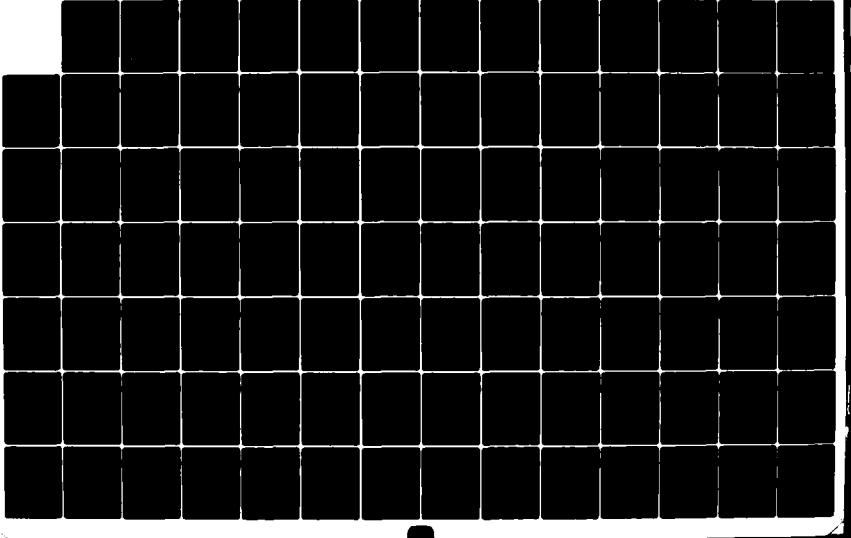
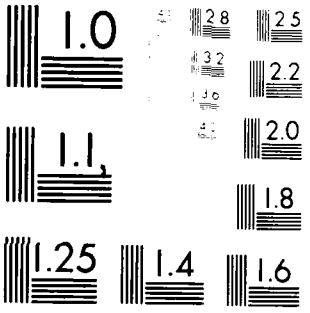
AD-A130 899 A SOFTWARE SCIENCE ANALYZER FOR COBOL REVISION(U) OHIO 1/2
STATE UNIV COLUMBUS COMPUTER AND INFORMATION SCIENCE
RES.. K C FUNG ET AL. 1982 OSU-CISRC-TR-83-2-REV
UNCLASSIFIED ARO-17150.3-EL DAAG29-80-K-0061 F/G 9/2 NL





(12)

TECHNICAL REPORT SERIES

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

OSU-CISRC-TR-83-2

A SOFTWARE SCIENCE ANALYZER FOR COBOL

K.C. FUNG

N.C. DEBNATH

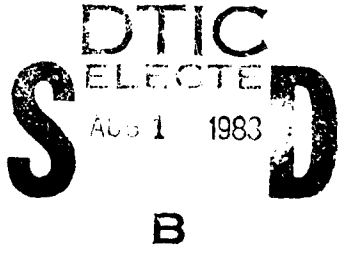
and

S.H. ZWEBEN

Research supported in part by

U. S. Army Research Office

Contract DAAG29-80-k-0061

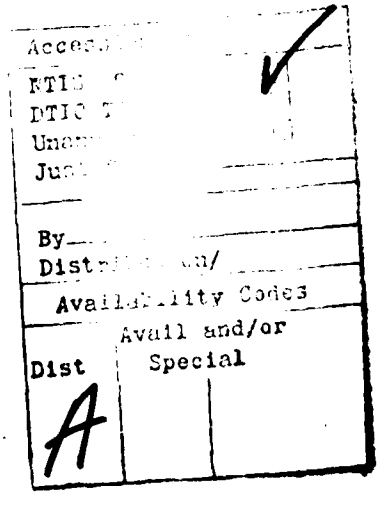


Computer and Information Science Research Center
The Ohio State University
Columbus, OH 43210

Revised Autumn 1982

ABSTRACT

An analyzer of COBOL programs which computes the metrics from software science is described. The report discusses the overall design of the analyzer, including detailed descriptions of each of its modules. It also contains instructions for the use and maintenance of the analyzer at Ohio State University.



PREFACE

This report is the result of research supported in part by the U. S. Army Research Office of Scientific Research under contract DAAG29-80-k-0061. It is being published by the Computer and Information Science Research Center (CISRC) of the Ohio State University in conjunction with the Department of Computer and Information Science. CISRC is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories.

INDEX TERMS

Software Engineering, Software Metrics, Software Science, COBOL

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

It is a major theory of software science (Halstead 77) that if we divide the basic elements of a given program into operators and operands according to a proposed counting strategy, the statistics of these operators and operands exhibit some interesting relationships to aspects of software quality. It is hopeful that these relationships can form a quantitative basis for the analysis of software. In order for this approach to gain widespread acceptance, it is necessary for these relationships to be validated on different classes of programs.

Halstead (Halstead 79), in a treatise of software science research, cites many studies which have sought to validate these relationships on programs written in many languages. Interestingly, though, none of these analyses involve COBOL! More recently, (Zweben and Fung 79) reported on the results of a preliminary study of COBOL programs which were counted manually. However, in order to gather large amounts of data on COBOL programs it is necessary to be able to count the operators and operands of COBOL programs mechanically. The computer program (or analyzer, as it acts almost like a lexical analyzer for COBOL) described in this report is the result of such an effort, undertaken at The Ohio State University, to streamline the counting process for the study of software science metrics.

It should be mentioned that a Software Science research group at Purdue University has developed another COBOL analyzer (Shen and Dunsmore 80). The Purdue University analyzer is written in COBOL whereas the Ohio State University (OSU) analyzer is written in PL/I. Both the analyzers are capable of handling Data as well as Procedure divisions of a COBOL program, as has been recommended by (Zweben & Fung 79). In addition, both programs allow users the option of providing their own definition of COBOL operators and operands. In particular, the OSU analyzer offers the added feature of context sensitive counting of various keywords, as will be discussed in this report.

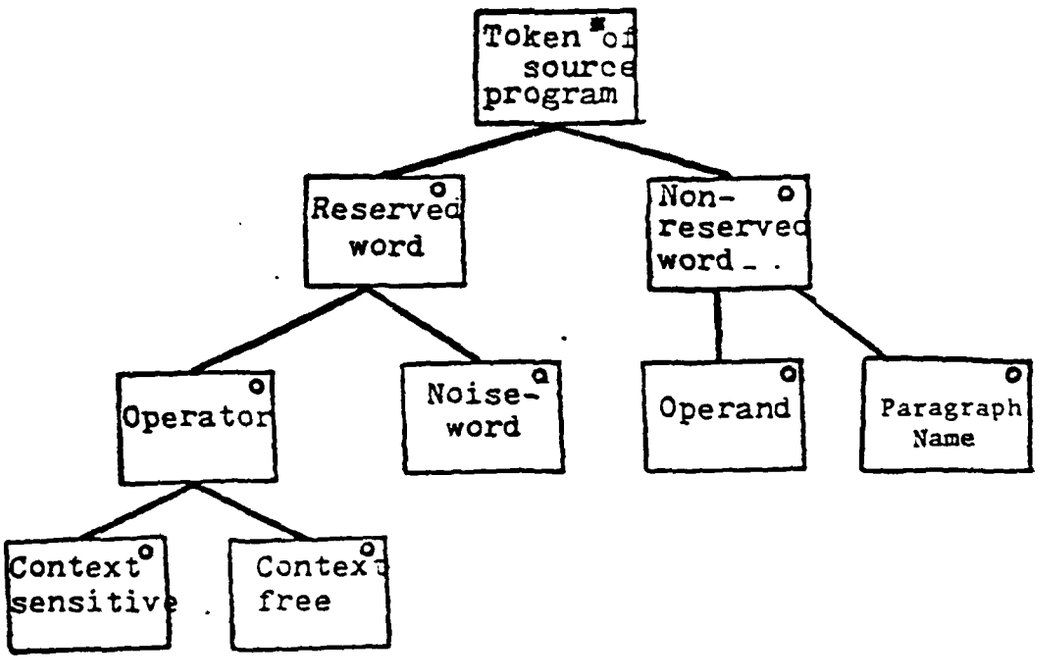
The next chapter describes the overall design of the OSU analyzer, and is intended for readers who may wish to study and/or modify the actual programming details. Chapter 3 provides the details of how to use the analyzer, either in "default" mode using a predefined counting strategy, or with user provided definition of COBOL operators and operands. A brief discussion about the future work to be done using this analyzer has been outlined in Chapter 4. For completeness of the report, four appendices are included. Appendix-A provides the explicit design document for the entire analyzer program. A detailed description of the existing files related to the analyzer is given in Appendix-B. In Appendix-C, a short procedure for maintaining the analyzer is mentioned. Finally, the procedure for invoking Purdue's analyzer at OSU has been explained in Appendix-D.

CHAPTER 2

OVERVIEW OF THE DESIGN OF THE COBOL ANALYER

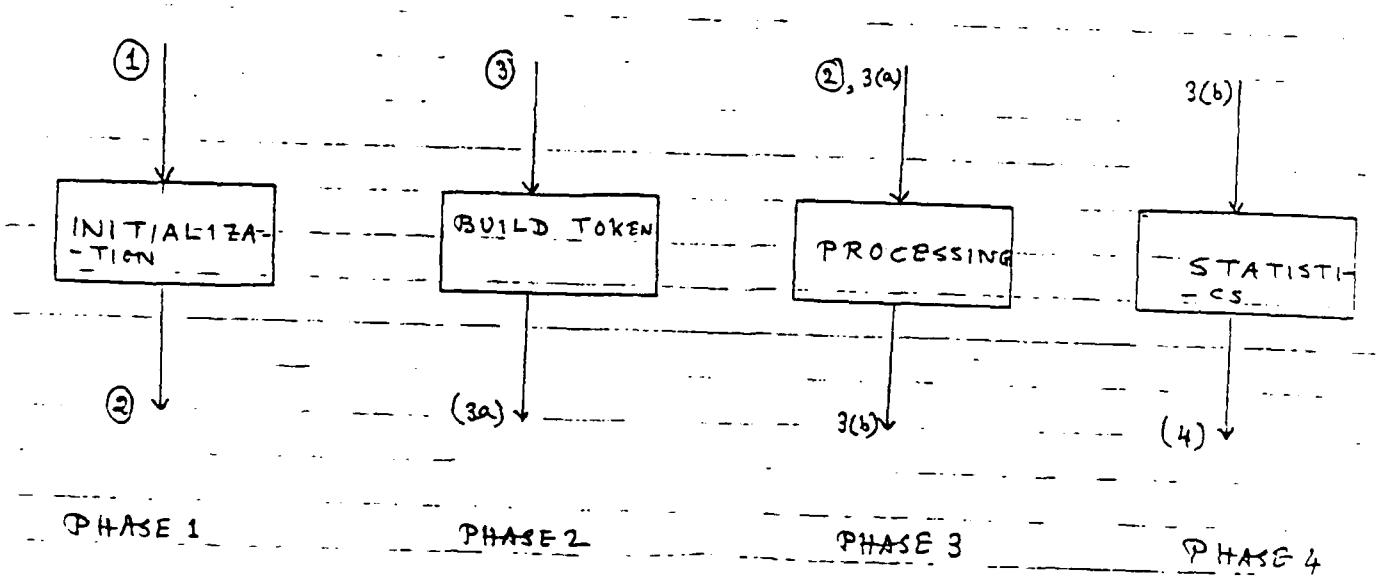
2.1: Program Structure

The strucure of the analyzer is based of the data structure of the token stream of a COBOL source program, which can be pictured as follows (using a Jackson design notation [Jackson, 75]):



Nonreserved words are programmer defined symbols and in general all nonreserved words except paragraph names are operands. Paragraph names are identified by their location (beginning in Col. 8) in a source language statement. Reserved words are language (compiler) defined symbols and they are usually well documented in the language manual. Most COBOL reserved words function as operators but some function as optional symbols to make the sentence structure more English like. In the usual counting strategy of software science, these optional symbols are ignored and are thus called noisewords. Some operators are context sensitive and additional actions are needed to identify and process them. Since reserved words can be defined precisely with the help of a language manual, operands (nonreserved words) of a given source program can be identified by checking against the list of reserved words.

The analyzer can be viewed as having the following four major phases.



The data structures used as input/output to each of the phases are the following.

1. Predefined operator and noiseword files.
2. List and Tree structures of operators/noisewords.
3. COBOL source program
   (a) Token list of the COBOL source program
   (b) Operator/Operand tree for the data and procedure divisions of the program.
4. Report of the Software Science metrics.

Initially, information about reserved words is used to build operator and noiseword trees (module INITIAL & INITIA2). This information consists of names of reserved words, alternative forms of these words (synonyms), and any context sensitive information concerning their use as operators. The input character stream of the source program is then broken down into a token stream (module BLDTKN & GETOKEN). Every token is first processed against the operator tree (module OPERATR & FILTER1). If it is not an operator, or if the operator tree provides no synonym or context sensitive information to determine how to deal with the token, it is processed against the noiseword tree (module NOISE & FILTER2). If it turns out to be none of the above, then the token is considered to be an operand and this information is entered into the operand tree (module OPERAND & FILTER3). After all the tokens are processed and the information concerning their classification is built into the appropriate trees, a module STAT is invoked to generate all the relevant statistics and the final output.

Note that two modules exist for each process other than the statistics generation. One module performs the process for the DATA division and the other performs the corresponding process for PROCEDURE division.

The program structure is best described by the following high level routine (used in the program though details have been deleted here for clarity).

```
INITIALIZE DATA STRUCTURES ;        /* Call INITIAL and INITIA2  */
DO WHILE (¬ END OF PROGRAM );
   CAPTURE A TOKEN ;                /* Call GETOKEN or BLDTKN  */
   COMPARE THE TOKEN AGAINST THE DEFAULT OPERATOR TREE ;  /* Call OPERATR or FILTER1
   IF THE TOKEN IS AN OPERATOR
   THEN
        PROCESS THE TOKEN AS OPERATOR ;
   ELSE
        COMPARE THE TOKEN AGAINST THE NOISE TREE ;        /* Call NOISE or FILTER2  */
   IF THE TOKEN IS (¬ OPERATOR AND ¬ NOISEWORD )
   THEN
        PROCESS THE TOKEN AS AN OPERAND ;                 /* Call OPERAND or FILTER3  */
END ;
PRODUCE THE ANALYZER REPORT ;     /* Call STAT  */
```

A characterization of the major data structures used by the program, a structure chart, and a description of major modules are given in the remaining sections.

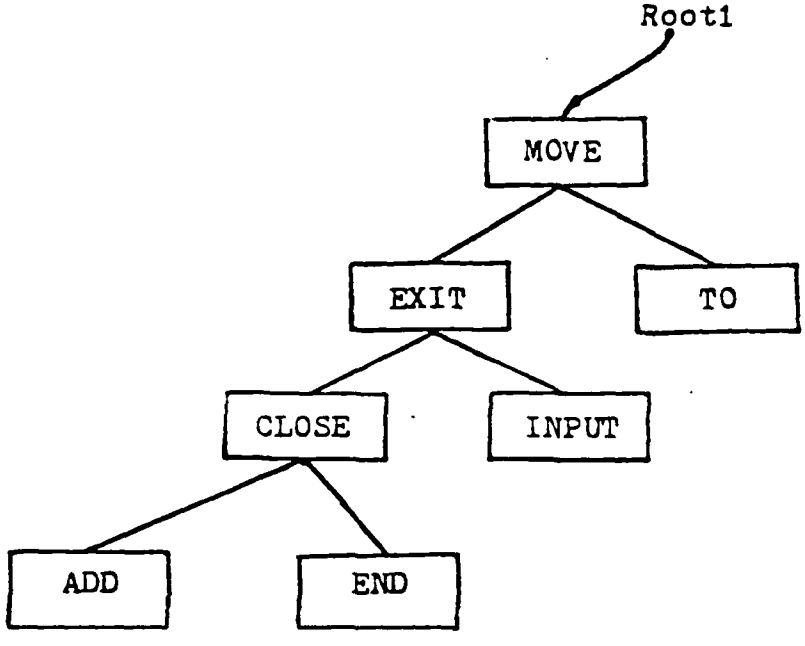
2.2 Internal Data Structures

Functionally there are six linked list structures in this program. One uses a binary tree structure, four use a linear list structure and the remaining one uses a circular linked list structure.

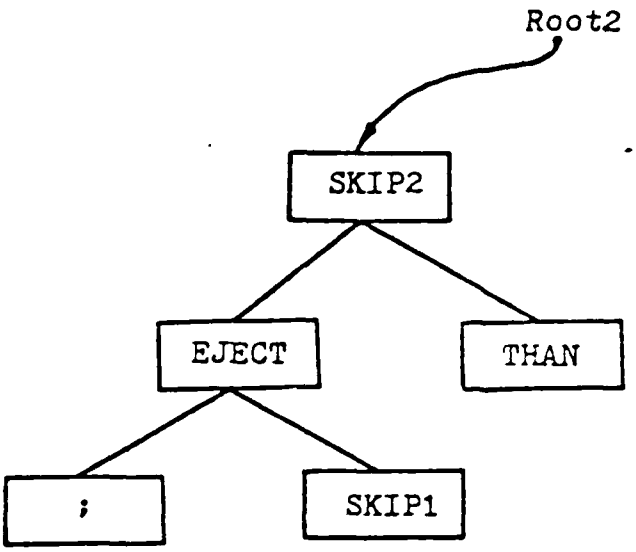
Binary Search Trees

There are three important examples of the binary tree node structure in the analyzer. These are the operator, noise, and operand trees.

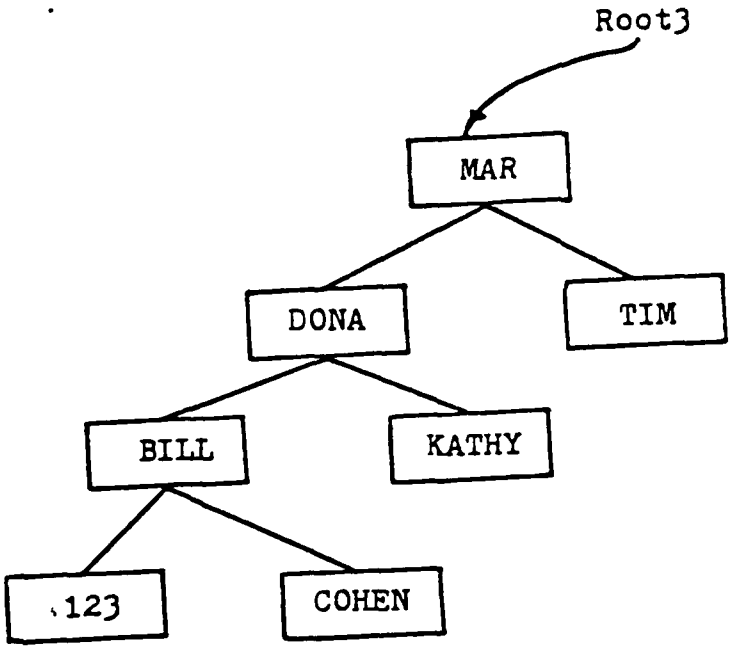
Operator tree



Noise tree

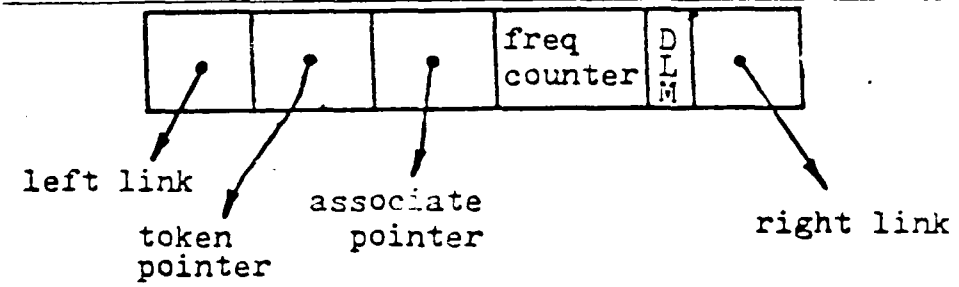


Operand tree



The nodes in a tree are related to each other in lexicographic order of the symbol with respect to the alphabet set used by the local computer (e.g. EBCDIC). Each node contains information about the frequency occurence of its token at the current point of the analysis (see the node structure below). After the trees are completed it is a simple matter to derive the software science measures ETA1, N1, ETA2, N2, and the frequency distribution of the operators and operands (see STAT under Section 2.4).

Structure of a node of the operator, noise, and operand trees



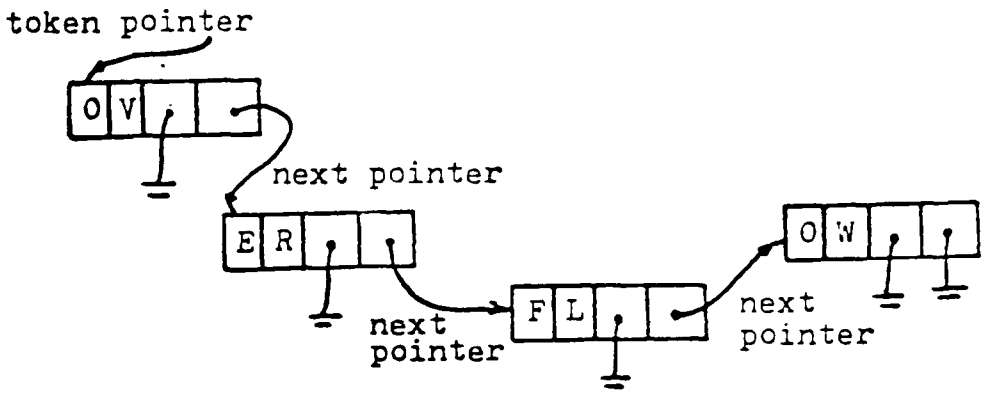
In the analyzer the PL/1 declaration for this structure is:

```
DCL 01 TREENODE          BASED (current),
       02 LPTR                 POINTER,
       02 RPTR                 POINTER,
       02 ASSOCIATE            POINTER,
       02 TKNPRT               POINTER,
       02 FREQCNTR             POINTER,
       02 DLMFLAG              BIT (1);
```

In general, left-link and right-link help to define the tree structure mentioned above; token pointer points to an internal list structure of the token associated with the node (token list described below); associate pointer leads to an associate list which contains context sensitive information about the token, if there is any; freq-counter contains the occurrence frequency of the token; and DLM (delimiter) is a flag used for a variety of purposes, one of which is to indicate whether a particular operator is a COBOL verb.

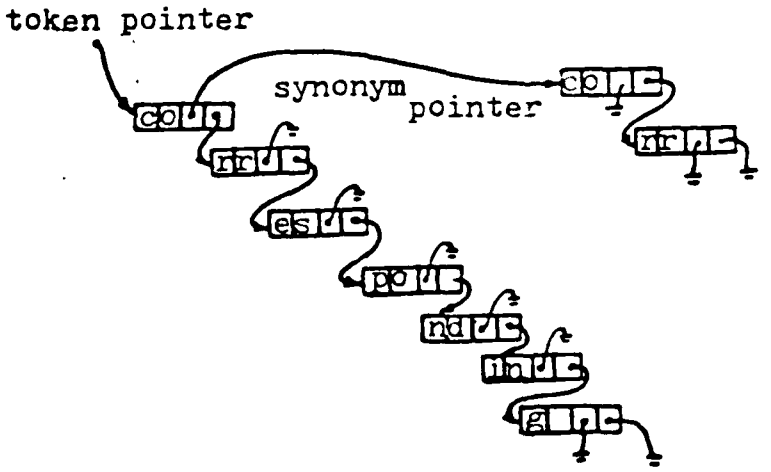
Token List

The tokens are represented internally by a linear list structure. Each node contains two adjacent characters of a token. In general a token of length x requires (x/2+1) nodes for its internal representation, and the token pointer of a tree node points to the first node of this list.



Synonym List

Synonymous tokens are linked up through the synonym pointer (3rd field) in the first node of their token list.

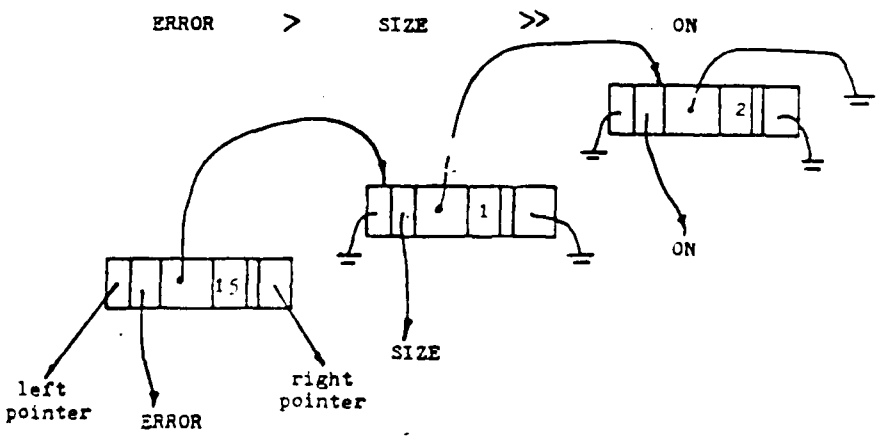


Associate List

The associate list is used to record context sensitive counting strategy rules for COBOL keyword tokens. These rules are given as "action pairs" in the input instruction list (see section 3.2 for the syntax and semantics of action pairs). A token affecting a previous token or being affected by a previous token has an associate list node linked to its tree node. The structure of an associate list node is similar to that of a tree node. If a token is sensitive to more than one previous token, additional associate list nodes are linked through the associate pointer field.

As an example, let us see how the data structure of an associate list corresponds to the description of the context sensitive relationship among the tokens `ERROR´, `ON´ and `SIZE´.

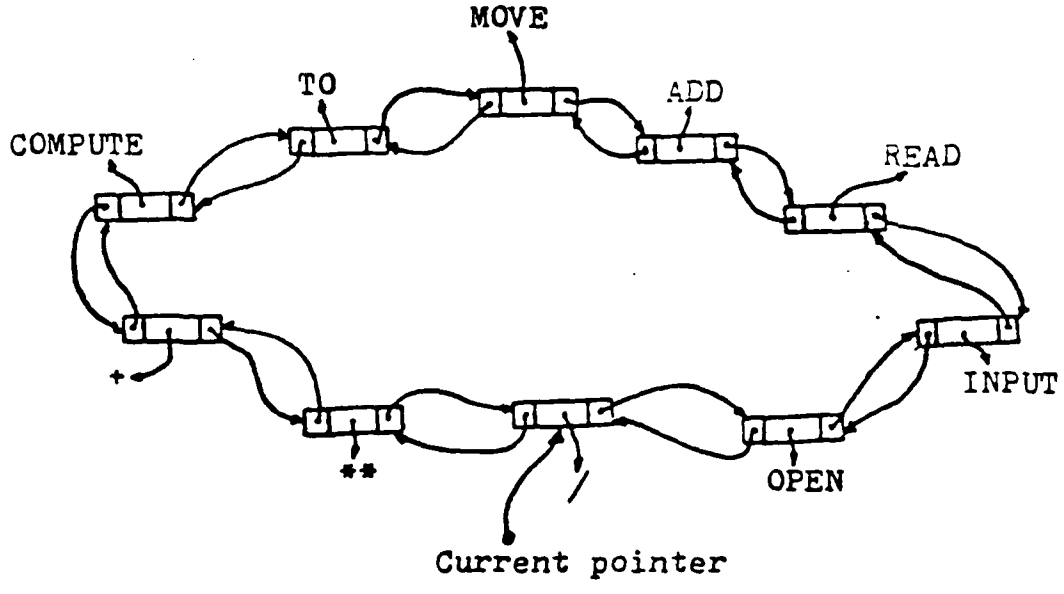
Our counting strategy suggests that in the context of `ON SIZE ERROR´, `ON´ and `SIZE´ should be counted as noisewords, and the token `ERROR´ representing this string should be counted as an operator. This can be expressed as the input instruction `ERROR > SIZE >> ON´ in the file OPER1 (see section 3.2 for a discussion of the syntax of these input instructions). To implement this instruction (which is an abbreviation of the two action pairs `ERROR > SIZE´ and `ERROR >> ON´) we build the associate list as follows.



The leftmost node in this example is a tree node for the token `ERROR´ and has a frequency count of 15 (arbitrarily chosen). The other two nodes in the example are not tree nodes, but are members of the associate list for the token `ERROR´. For the `SIZE´ node, the 1 in the frequency field denotes that in the case of `ERROR´ following `SIZE´ by one token, `SIZE´ is considered as a noiseword (positive value associate with >). The 2 in the frequency field of `ON´ denotes that in case of `ERROR´ following `ON´ by two tokens, `ON´ should be considered as the noiseword.

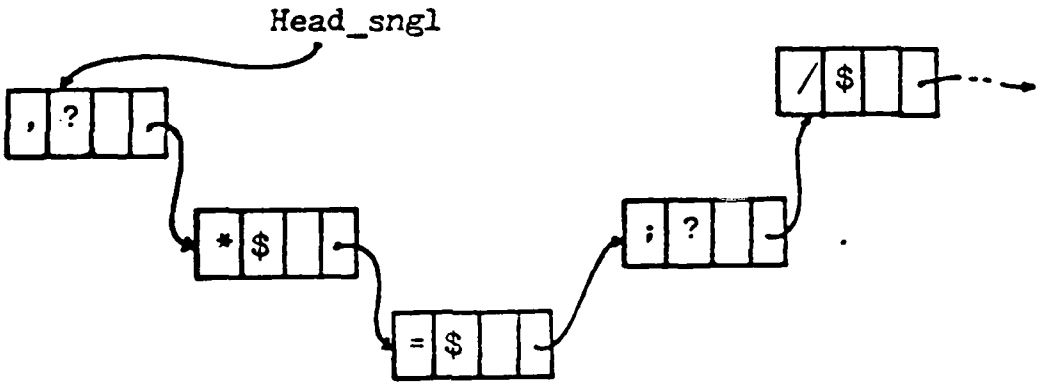
Historical List

This is a circular list which saves the last ten tokens for backpatching the frequency count of any context sensitive token.



Special Character List

This linear list links up all special break-characters which are also considered as tokens. This list is consulted by every incoming character of the source program. The second character field of the token node contains a `$' or `?' depending on whether the character is an operator or noiseword. This list built as the (user defined or default) operator and noiseword files are read into the analyer. Single character tokens from these files are special break characters.



2.3: STRUCTURE CHART

The folowing figures show the complete modular structure of the Analyzer.

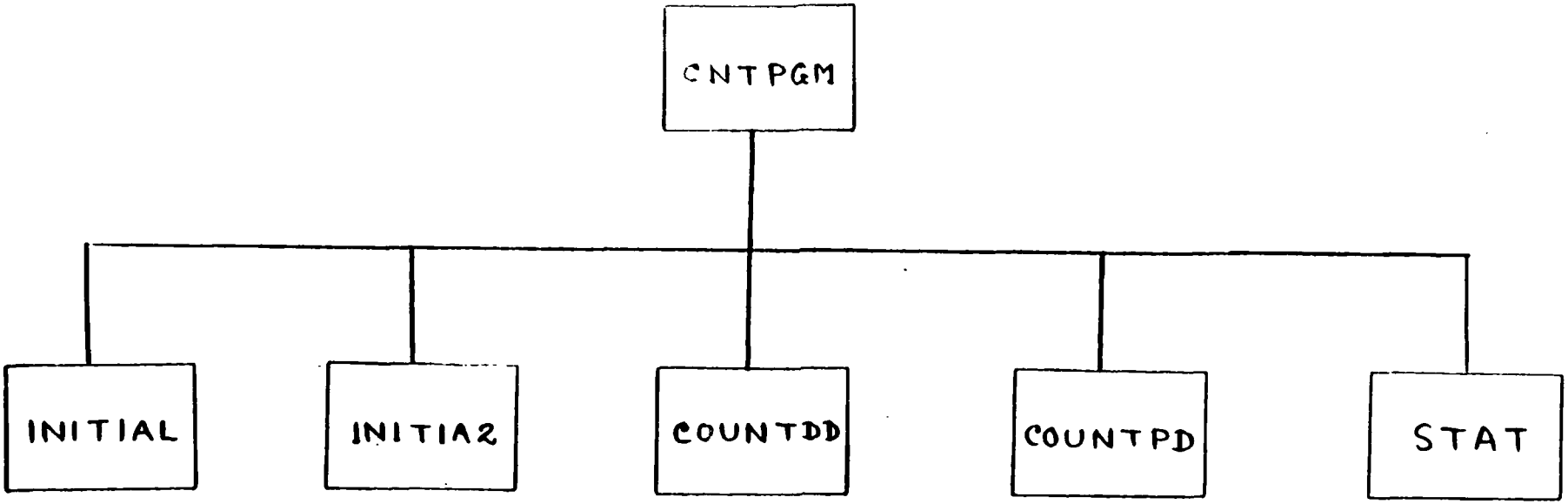
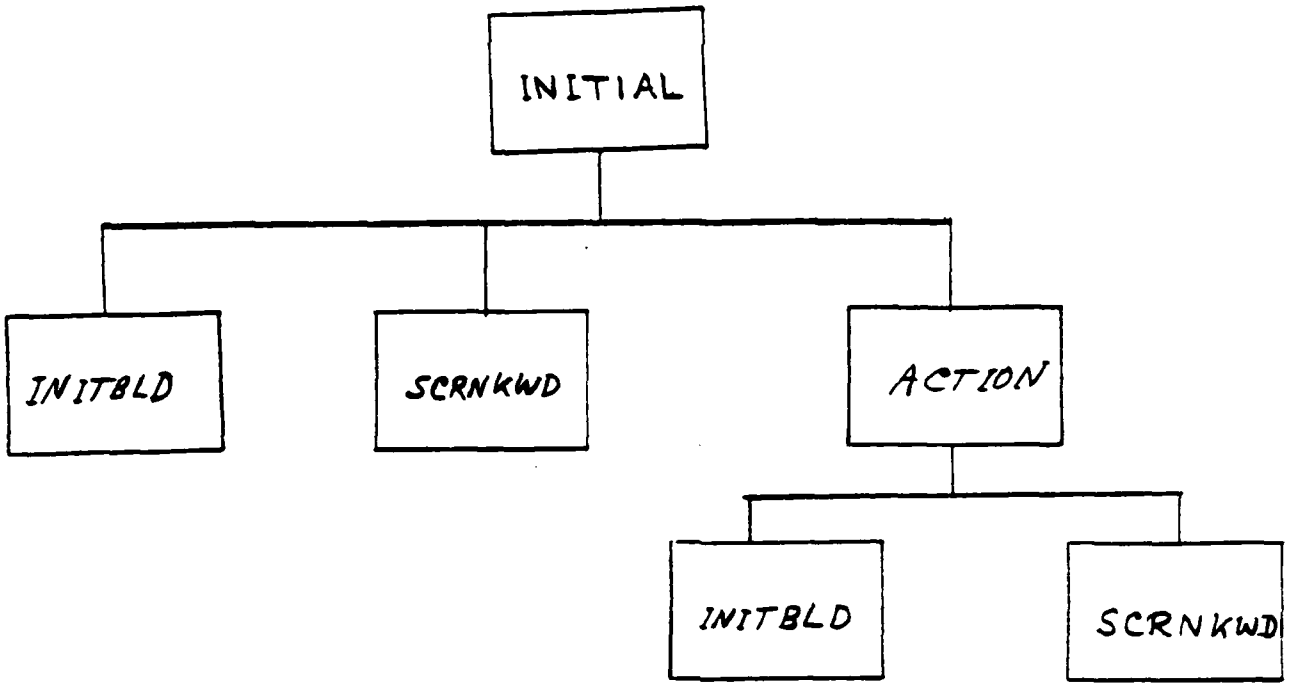


Fig. 2.3b | Fig. 2.3b | Fig. 2.3c | Fig. 2.3d | Fig. 2.3e

Fig. 2.3a

Note: All unshaded boxes are external subroutines and some of these serve as primitive subroutines e.g. SURCH, INSERT, FREETKN, COMPARE, PRNTTKN, INITBLD, and SCRNKWD.



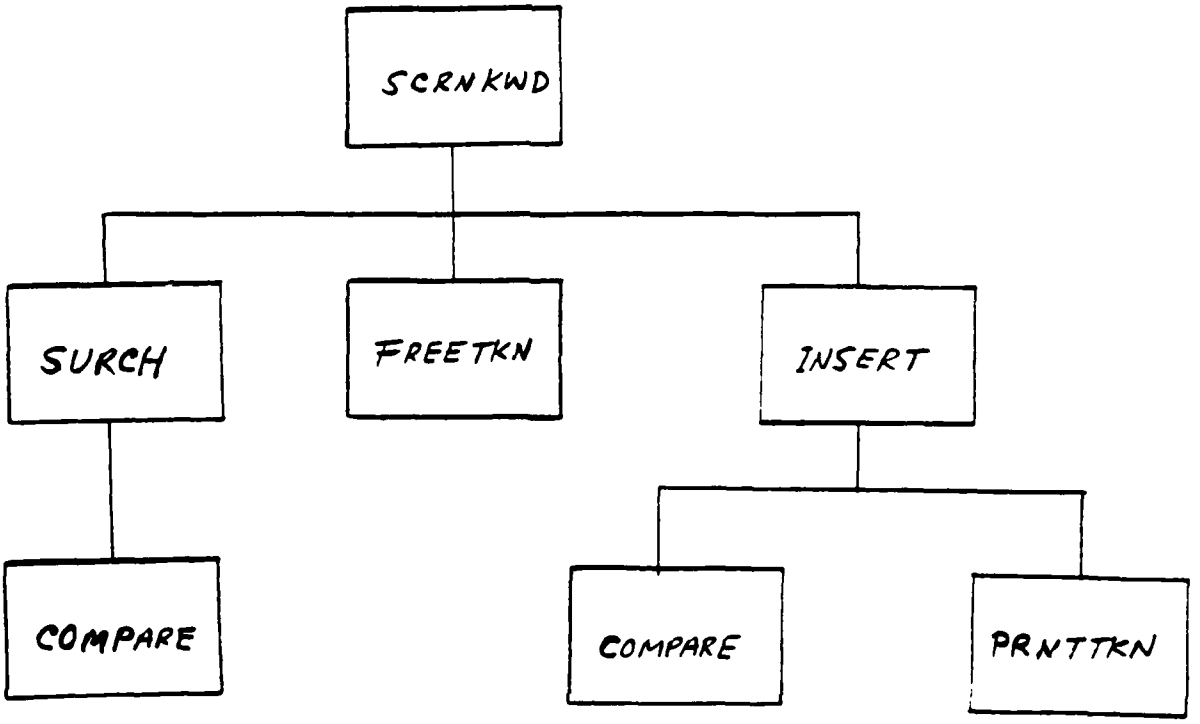


Fig. 2.3b

Note: INITIA2 has the same structure and subroutines as INITIAL.

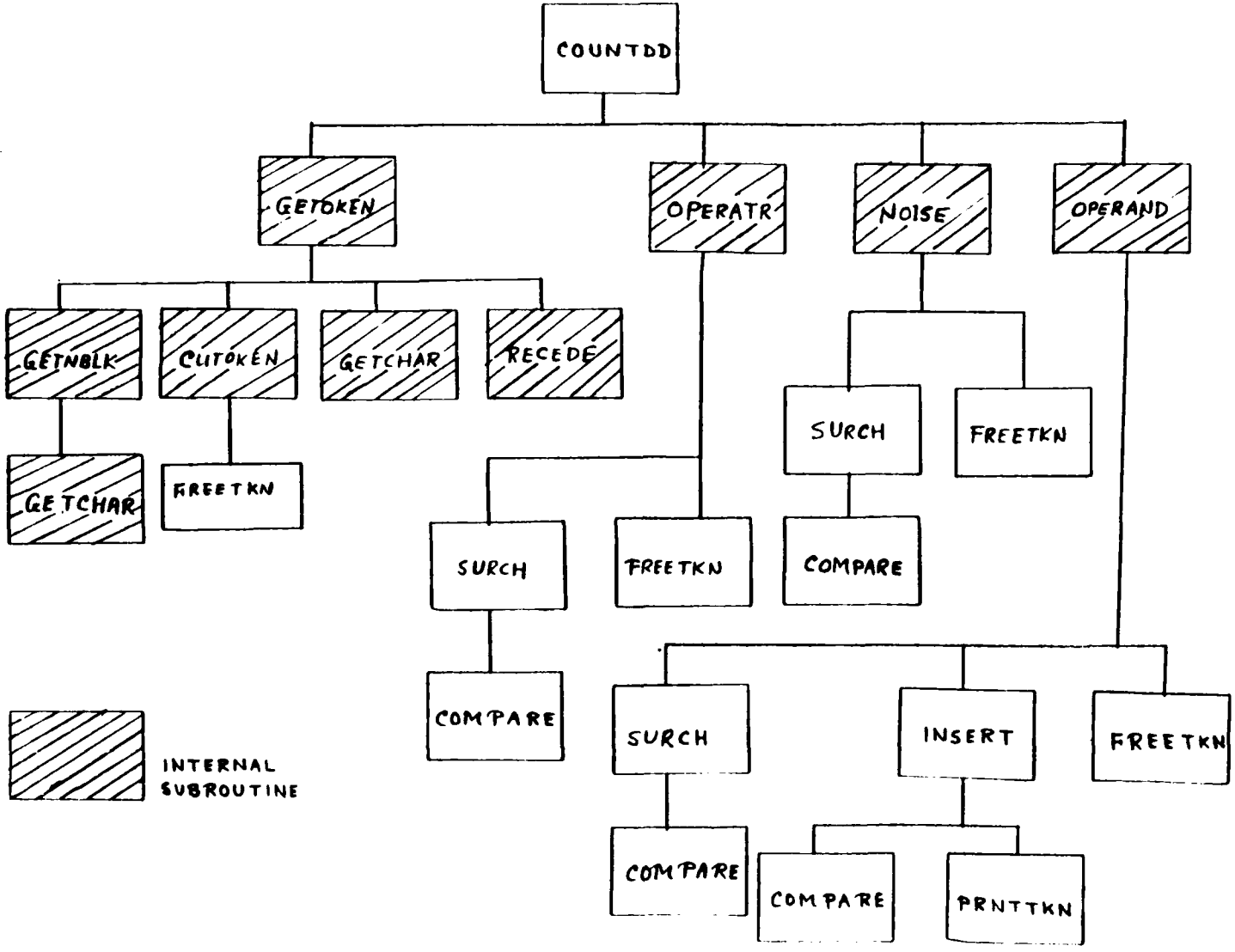

COUNTDD
GETOKEN
OPERATR
NOISE
OPERAND
GETNBLK
CLITOKEN
GETCHAR
RECEDE
SURCH
FREETKN
GETCHAR
FREETKN
SURCH
FREETKN
COMPARE
COMPARE
SURCH
INSERT
FREETKN
COMPARE
COMPARE
PRNTTKN
INTERNAL SUBROUTINE

Fig. 2.3c

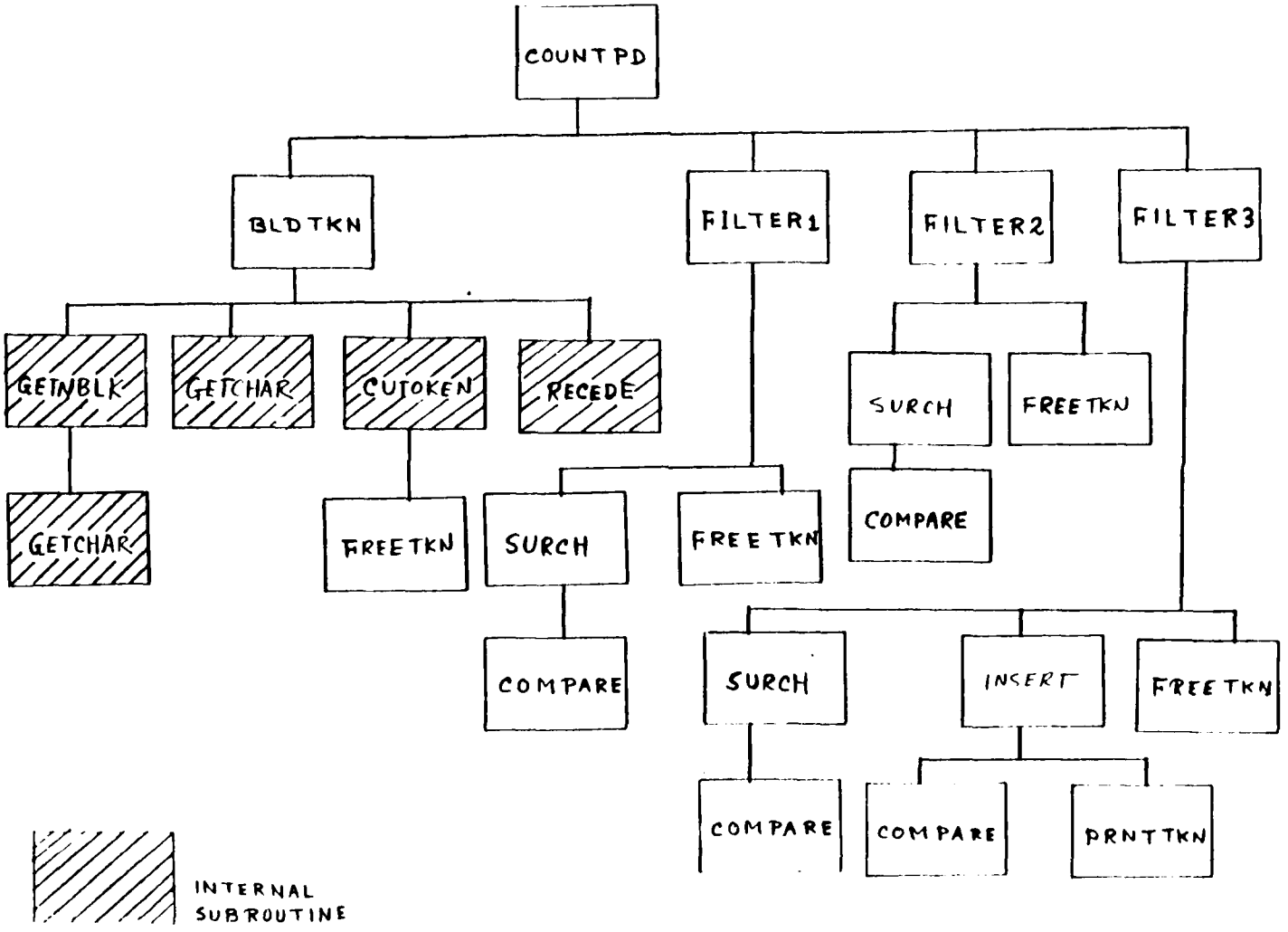

COUNTPD
BLDTKN
FILTER1
FILTER2
FILTER3
GETNBLK
GETCHAR
CUTOKEN
RECEDE
SURCH
FREETKN
GETCHAR
FREETKN
SURCH
FREETKN
COMPARE
COMPARE
SURCH
INSERT
FREETKN
COMPARE
COMPARE
PRNTTKN
INTERNAL SUBROUTINE

Fig. 2.3d

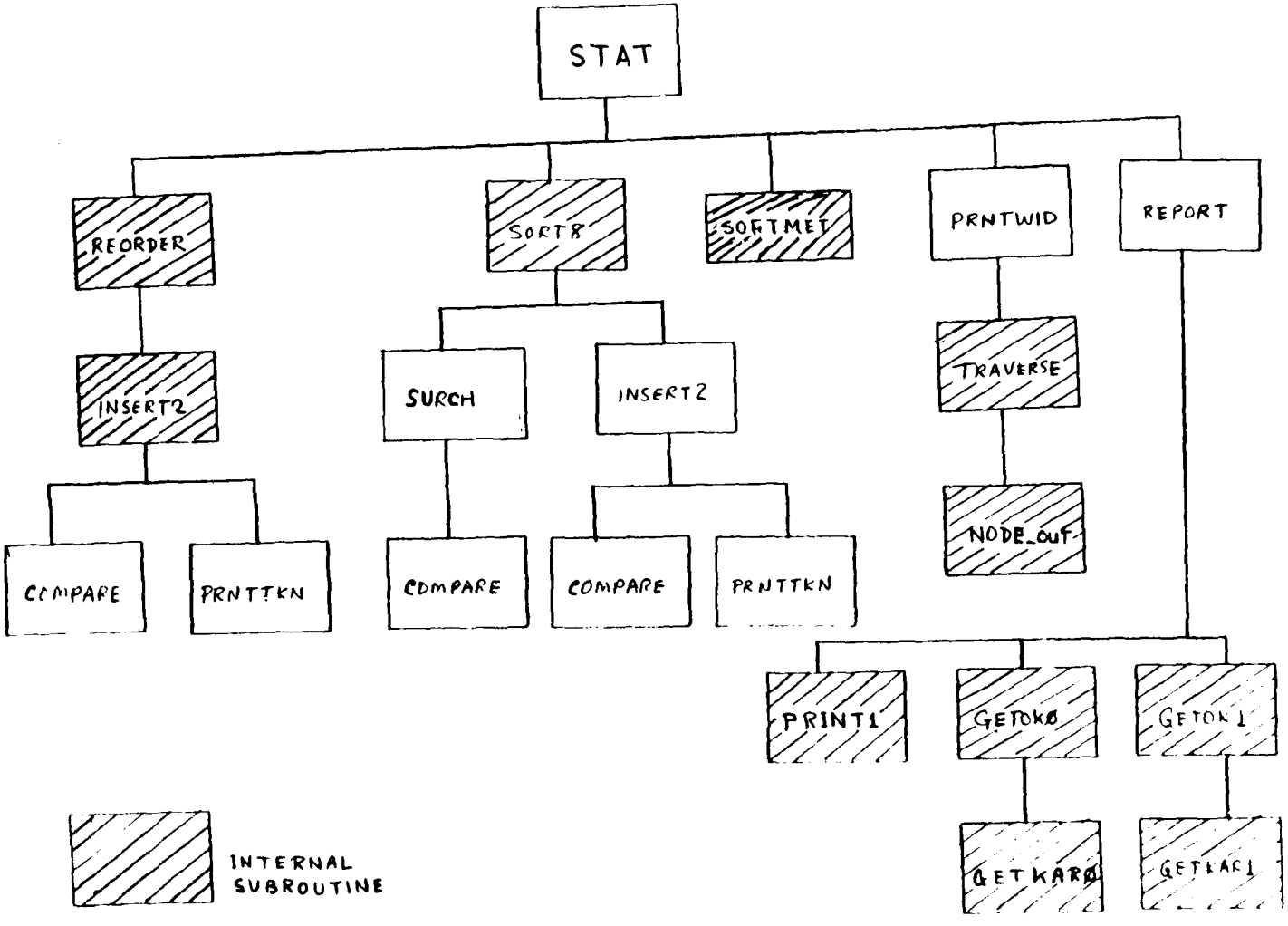

STAT
REORDER
SORTR
SOFTMET
PRNTWID
REPORT
INSERT2
SURCH
INSERT2
TRAVERSE
COMPARE
PRNTTKN
COMPARE
COMPARE
PRNTTKN
NODE_OUT
PRINT1
GETOKØ
GETOK1
GETKARØ
GETKAR1
INTERNAL SUBROUTINE

Fig. 2.3e

2.4 Description of Major Modules*:

INITAL (INITIA2): Initializes the fundamental data structures needed for analyzing the Procedure (Data) division of a COBOL program. In particular, this module builds operator and noiseword trees according to the input instructions supplied by user or the default file descriptions. It also constructs the token, associate and synonym lists.

INITBLD: Captures a token from the input instruction stream and builds a linked list structure for the token.

SCRNKWD: Screens the keyword (token) captured by INITBLD. If it is a new member of the operator/noiseword tree then it is inserted into the appropriate tree. Otherwise the linked list structure of the token is freed.

ACTION: Processes the action pairs (see Section 3.2) from the input instructions. These action pairs define synonyms and context sensitive information.

COUNTDD (COUNTPD): Scans the Data (Procedure) division of a COBOL program. Each occurrence of a token is classified on the basis of the counting strategy defined in INITIA2 (INITIAL), and the frequency count of each unique operator or operand is updated.

GETNBLK: Reads an input record of the COBOL source program starting from column 8 and returns the first nonblank character in the record.

* A more thorough description of each module is contained in Appendix-A.

GETOKEN (BLDTOKEN): Scans the records in Data (Procedure) division of a COBOL source program, and captures tokens from the input character stream. It also builds the linked list structure for the tokens.

This routine is more powerful than INITBLD in that it can capture a literal string as a token and it can determine the function of a variety of break-characters. For example, `-' may be used as a hyphen or as the subtraction operator, and `.' may be treated as a decimal point in a real number or as a delimiter.

OPERATR (FILTER1): Processes the incoming token in Data (Procedure) division against the operator tree of Data (Procedure) division.

If a match is found in the operator tree then increments its frequency count and frees the storage of the incoming token. Otherwise passes the token to NOISE (FILTER2).

NOISE (FILTER2): Checks the incoming token against the noiseword tree of the Data (Procedure) division. If a match is realized in the noise tree then frees the current token. Otherwise passes it down to next phase, namely OPERAND (FILTER3).

OPERAND (FILTER3): The incoming token must be an operand by default. This module searches the operand tree of Data (Procedure) division in order to see if a match exists. If a match is found then increments the frequency count of the current token and frees the token. Otherwise inserts the new token into the operand tree of the Data (Procedure) division and updates its frequency count.

STAT: Produces a report of the analysis. STAT makes use of the operator and the operand trees developed during the early stages. In traversing the operator tree, the number of unique operators (ETA1) is obtained by counting the tree nodes having nonzero frequency. The total number of operands (N1) is found by adding up the values of all the frequency counter fields. Similar treatment is followed for operand tree in order to find the number of unique operands and also the total number of operands (i.e. ETA2 and N2 respectively). This module also produces the frequency distribution of all the operators and operands in different divisions of a COBOL program in a sorted order.

REORDER: Sorts the operator and operand trees of the Procedure division as well as the operator tree of the Data division in order of frequency counts. It also provides information about ETA1, N1, ETA2, N2 and the number of statements in the appropriate division.

SORT8: This module is a slightly modified version of REORDER, and is used to sort the operand tree of the Data division. In addition to the number of unique operands and the total number of operands in the Data division, it also calculates the number of common operands between the operand trees of the Data and Procedure divisions. This number of common elements is used to find the number of unique operands in the entire program, since the number of unique operands in the whole program = (sum of the operands in both Data and Procedure divisons) - (number of common operands between the Data and Procedure divisions).

SOFTMET: Produces the final values of all Software Science metrics for Data and Procedure divisions.

PRNTWID: This module generates separate files containing the frequency distribution of all the tokens in both Data and Procedure division. The file is broken down into 80 character records, and is composed of node units. Each node unit corresponds to a token and consists of a 3-byte node length, a 4-byte frequency count, and the token symbol of up to 256 characters.

REPORT: Reads three different files, namely, SYSUT0, SYSUT1 and SYSUT2 generated by PRNTWID and STAT. SYSUT0, SYSUT1 and SYSUT2 contain all information from Data division, Procedure division and the program, respectively, necessary for producing the desired output. It generates a report of Software Science metrics. The frequency distribution of tokens in Data and Procedure divisions are displayed in parallel. Summary accounts of Data division, Procedure division, and the entire program are displayed at the end (see section 3.3).

CHAPTER 3

USE OF THE COBOL ANALYZER

In order to obtain Software Science measures, a set of unambiguous rules which defines the partition of symbols into operators and operands in a language is required. Different authors may come up with slightly different rules for the same language, even if the same language compiler is being used. The discrepancies are often due to disagreement of interpretation of the definitions of operator and operand given in (Halstead 77). An operand is defined as a "variable or constant" in a program. An operator is defined as "a symbol or combination of symbols that affect the value or ordering of an operand" in a program.

To allow for changes in the counting procedure by different users, the OSU analyzer allows the option of invoking a predefined set of rules or defining a new set of rules. Section 3.1 describes the former option, and Section 3.2 discusses the latter.

3.1 Default Mode

The counting strategy used in this analyzer considers entries in both Data and Procedure Divisions. It is governed by the following rules:

1. OPERANDS (Data and Procedure divisions).

Any reference to a distinct operand is counted as an occurrence of that operand. An operand is any of the following:

a. A file-name, e.g., CARD-INPUT-FILE.

b. An identifier, e.g., EMPLOYEE-NUMBER.

c. A literal, e.g., `BILL´ or 1234.

A paragraph name or section name is not considered as an operand. Together with PERFORM or GO TO it is considered as an operator.

2. OPERATORS (Procedure Division).

Any reference to a distinct operator is counted as an occurrence of that operator. An operator is any of the following:

a. A logical operator, e.g., OR, AND.

b. A relational operator, e.g., =, EQUAL, LESS THAN.

c. An arithmetic operator, e.g.,+,-,* and /.

d. A key word or required word in a valid COBOL statement with the exception of GO TO, PERFORM, CALL and ALTER. Any group of keywords functioning as an operator is counted as a single operator. Examples of such keywords or keyword combinations are: IF, ELSE, NEXT SENTENCE, UNTIL, AT END, READ, and OPEN.

e. Noisewords are not considered as operators, and are ignored in the counting.

f. A transfer of control: Any transfer of control to a paragraph name, section name or subprogram name is counted an occurrence of the operator associated with that name, e.g., GO TO paragraph-1, GO TO paragraph-2, and PERFORM paragraph-1 are all distinct operators.

g. A parenthesis pair e.g., A - (B - C) has one occurrence of the operator denoted as `parenthesis pair´.

3. OPERATORS (Data Division).

All keywords/required words in a valid COBOL statement are considered as operators e.g. FD, BLOCK, VALUE, REDEFINES, PICTURE etc.

4. Each occurrence of a COBOL verb adds one to the count of the operator denoted as, ´end of statement´.

5. Periods, commas and semi-colons are not counted.

Complete lists of the operators and noisewords for both Data and Procedure divisions, based on ANSI COBOL 1973 and including language extensions from IBM OS Version 4, are shown in appendix-B. The order of the records is so as to create well balanced tree structures for the analyzer (see Chapter 2 for details concerning these structures), with an effort to put more frequently occurring entities near the root of the tree.

To run a job at Ohio State's Instruction and Research Computer Center (IRCC) using default mode, the following sequence of inputs is required:

```
//jobname  JOB  ...account number...
//PROCLIB  DD  DSN=TS0618.PROCLIB,DISP=SHR
//    EXEC OSUCNTPM
//SOURCE  DD *
        .
        .
        .
  (source program)
        .
        .
        .
/*
//
```

The JCL file , OSUCNTPM, used to run the analyzer is listed in Appendix-B.

It should be noted that currently the students of different COBOL courses at OSU use WIDJET and WYLBUR on-line systems to run their jobs on the AMDAHL 470. WIDJET and WYLBUR users utilize the following set of JCL to run the analyzer.

```
// JOB
/*JOBPARM V=D
//PROCLIB DD DSN = TS0618.PROCLIB
// EXEC INTERNAL
/PROGRAM DD *
$JOB  xxxxxx  student-name
    .
    .
    .
    .
    COBOL SOURCE PROGRAM
    .
    .
    .
    .
$ENTRY
//
```

Where xxxxxx corresponds to 2-digit LAB-ID and 4- digit AUTHOR-ID.

Unlike `OSUCNTPM´, this JCL allows the analyzer to gather outputs into a separate disk file (see Appendix-C). The detailed structure of INTERAL and other necessary JCL invoked by INTERNAL are given in Appendix-B.

It is worth mentioning that recently an EXEC program has been developed which allows students to run the analyzer more conveniently, while keeping the analyzer secure from student modification. The entire EXEC program listing is also included in Appendix-B for completeness. Because of the present facility, the students need to use only a simple command (called ANALYZE) to run their program through the analyzer instead of using the JCL given above.

3.2 User Defined Operators and Operands

It is also possible to run the analyzer with user defined operators and operands. When using the analyzer in this mode three input files are required. They are each discussed below.

Source Program

DDname: SOURCE

This file contains the COBOL program to be analyzed. Each record is 80 bytes long and corresponds to a line of the source program. The size of the source program is not limited by the analyzer but by the memory available because operand storage is dynamically allocated and freed. Currently only one source program can be analyzed in one execution.

Operator File

DDname:OPER

This file contains the definition of operator for the counting strategy. Each record is 80 bytes long.
The syntax of an operator file is:

```
[.] keyword-1 [relation keyword-2].....
```

The period before keyword-1 is optional. Its occurrence indicates that keyword-1 should be considered as a COBOL verb. Since a COBOL statement is delimited by the verb of the next statement, counting the verbs of COBOL is an indirect way to count the number of statements in the source program.

Keyword-1 is a symbol to be considered as an operator in the counting strategy. If keyword-1 has been registered before, it is not registered again.

The `relation keyword-2' pair is optional. Its presence indicates that a certain action is to be performed in the context of keyword-1 and keyword-2. Multiple action pairs signifies multiple actions on keyword-1. The kind of action to take place is defined by the relation of the instruction as follows:

`=' means that keyword-2 is to be considered as a synonym of keyword-1. They are to be treated as equivalent tokens in the counting strategy.

`>' means that on encountering keyword-1, if the most recent token is keyword-2, keyword-2 is to be considered as a noiseword.

`<' means that on encountering keyword-1, if the most recent token is keyword-2, keyword-1 is to be considered as a noiseword.

`>>' means that on encountering keyword-1, if the second most recent token is keyword-2, then keyword-2 is to be considered as a noiseword. Up to six multiple `>' may be used, implying that backtracking by six operators is possible.

`<<' means that on encountering keyword-1, if the second most recent token is keyword-2, then keyword-1 is to be considered as a noiseword. Up to six multiple `<' may be used.

One easy way to recognize which keyword is to be considered as a noiseword is to look at the `point´ of the relation. The relational operator always points at the noiseword.

e.g. Keyword-1 >>>> keyword-2 << keyword-3

The first relation points at keyword-2; thus keyword-2 is to be considered as a noiseword. But the second relation points at keyword-1; thus in the context of keyword-1 and keyword-3, keyword-1 is to be considered as a noiseword. Note that this input record does <u>not</u> define any relationship between keyword-2 and keyword-3.

By entering into the operation file a list of relation - keyword pairs, the user of this analyzer may define the operators and operands of his counting strategy based on information obtained from the language manual supplied by the vendor.

Example 1

Different versions of a COBOL compiler may have different repertoires of reserved words. The counting program used under different COBOL compilers must reflect this variation through entries into the operator file. For example, reserved words that begin with the letter `b´ under two popular compilers have the following difference.

1974 ANSI COBOL	1973 IBM OS Version 4
before	basis
blank	before
block	beginning
bottom	blank
by	block
	bottom
	by

In switching from ANSI COBOL to the compiler for IBM Version 4, two keywords, `basis´ and `beginning´, are to be added into the operator file. That is, suppose the entries for reserved words that begin with the letter `b´ under a 1974 ANSI/COBOL compiler are (n cards existing before these entries):

```
                              111111111
              column    123456789012345678...
                        .
                        .
                        .
  Card n+1              BEFORE
  Card n+2              BLANK
  Card n+3              BLOCK
  Card n+4              BOTTOM
  Card n+5              BY
```

Then the entries under a 1973 IBM OS Version 4 compiler become:

```
                              111111111
              column    123456789012345678...
                        .
                        .
                        .
  Card n+1              BEFORE
  Card n+2              BLANK
  Card n+3              BLOCK
  Card n+4              BOTTOM
  Card n+5              BY
  Card n+6              BASIS
  Card n+7              BEGINNING
```

Example 2

According to the DEC-10 Version 4 COBOL compiler, the use of the EXAMINE verb should follow the general format:

```
                                        ALL
EXAMINE  identifier  TALLYING  LEADING      Literal-1
                                UNTIL FIRST
    [REPLACING BY literal-2]
```

Underlined capitalized words are considered as keywords. Capitalized words which are not underlined are considered noisewords. To define lexical units according to the above general format the following cards are entered into the operator file.

```
.EXAMINE
ALL < TALLYING
FIRST < UNTIL
BY < REPLACING
```

The period before `EXAMINE´ instructs the analyzer to consider every occurrence of `EXAMINE´ as an occurrence of an operator as well as an occurrence of the operator ` end of statement´.

`ALL < TALLYING´ instructs the analyzer to consider `ALL´ as a noiseword if its preceding word is `TALLYING´; otherwise both `ALL´ and `TALLYING´ are to be considered as operators.

`LEADING´ instructs the analyzer to register `LEADING´ as an operator. However, since there is no period preceding it, no occurrence of the `end of statement´ operator is registered.

`FIRST < UNTIL´ and `BY < REPLACING´ are instructions which enable the keyword pointed at to be considered as operator or noiseword according to context, similar to `ALL < TALLYING´.

Example 3

According to the following general format,

```
        CORRESPONDING
ADD                      Identifier-1 TO identifier-2
        CORR

        [ROUNDED] [ON SIZE ERROR imperative-statement]
```

the operator file should include the following cards:

```
.ADD
CORRESPONDING = CORR
TO
ROUNDED
ERROR > SIZE >> ON
```

The period before `ADD´ is to instruct the analyzer to register `ADD´ as a COBOL verb. Thus every occurrence of `ADD´ increments the count of both `ADD´ and `end of statement´.

`CORRESPONDING = CORR´ instructs the analyzer to consider `CORRESPONDING´ and `CORR´ as equivalent tokens.

`TO´ and `ROUNDED´ instructs the analyzer to register these two symbols as operators.

`ERROR > SIZE >> ON´ instructs the analyzer to ignore `SIZE´ if on encountering `ERROR´ the most recent token is `SIZE´, and to ignore `ON´ if on encountering `ERROR´ its second most recent token is `ON´.

Noiseword file

DDname:NOISE

This file contains all the individual noisewords which are not considered as operators or operands at any time during the analysis. Each individual noiseword is placed on a separate input record (80 bytes long). Some noisewords are sensitive to context and are considered as operators only in certain situations. For example, `TO´ is a noiseword in `EQUAL TO´ but is an operator in `MOVE X TO Y.´ Information of this kind should be supplied by the user to the operator file (see above examples). The noiseword file only contains symbols which are always considered to be noisewords. This set of `true´ noisewords has to be determined before analysis in order to define the operands of the counting strategy. Any token which is not contained in the operator file or the noiseword file will be categorized as an operand.

The following is the JCL required to run the analyzer at Ohio State University´s Instruction and Research Computer Center if user defined operators and operands are employed:

```
//jobname  JOB   ...account number...
//    EXEC  PGM=OSUCNTPM
//STEPLIB  DD  DSN=TS0618.LOADLIB,DISP=SHR
//OPER  DD*
     .
     .
     .
  (operator file)
     .
     .
     .
/*
//NOISE  DD*
     .
     .
     .
  (noiseword file)
     .
     .
     .
/*
//SOURCE  DD*
     .
     .
     .
  (source program)
     .
     .
     .
/*
//SYSPRINT  DD  SYSOUT=A
//
```

3.3 Output of the Analyzer

The following outputs are generated by the analyzer:

- Echoes of the operator file and noiseword file
- List of tokens scanned during the analysis
- Frequency distribution of operators and operands
- Number of unique operators (ETA1)
- Number of operator occurrences (N1)
- Number of unique operands (ETA2)
- Number of operand occurrences (N2)
- Vocabulary (ETA)
- Program Length (N)
- Estimated program Length (NH)
- Total Number of Statements (NOS)

  Program Volume (V)
- Program Level (LH)
- Language Level (LAMBDA)
- Intelligence Content (INTELL)
- Programming Effort (EFFORT)

A sample of the actual output format produced by the analyzer for the frequency distributions and metrics summary follows.

SOFTWARE SCIENCE METRICS REPORT

PROGRAM-ID : 02
AUTHOR-ID : 0470
DATE : 01/14/83

DATA DIVISION OPERATOR	FREQUENCY
BLOCK	2
STANDARD	3
.	.
.	.
.	.
VALUE	45
PICTURE , PIC	123
.	150

PROCEDURE DIVISION OPERATOR	FREQUENCY
CLOSE	1
OPEN	1
.	.
.	.
.	.
WRITE	12
MOVE	26
TO	26
/*EOS*/	65

DATA DIVISION OPERAND	FREQUENCY
$$$$$9.99	1
EOF-FLAG	1
FIRST-LINE	1
HASH-NUM	1
.	.
.	.
.	.
ZEROS	7
SPACES	13
FILLER	32
10	44
05	51

PROCEDURE DIVISION OPERAND	FREQUENCY
'NO'	1
HASH-VALUE	1
LINE-TWO	1
EOF-FLAG	3
.	.
.	.
.	.
UPDATE-FILE	6
1	7
3	9
PRINTER-RECORD	12

	ETA1	N1	ETA2	N2	ETA	N	NII	NOS	V	LH	LAMBDA	INTELL	EFFORT
DATA DIVISION	10	284	178	395	188	678	1363	123	5128	0.0901	41.63	462.1	56825
PROC DIVISION	28	208	74	131	102	340	594	65	2268	0.0403	3.68	91.4	56277
PROGRAM	38	492	187	526	225	1018	1610	188	7961	0.0187	2.78	148.8	425775

CHAPTER 4

SUMMARY

The present form of the analyzer herein, developed by the software metrics research group at Ohio State University, handles operators and operands in both Data and Procedure divisions of a COBOL program. The availability of this analyzer makes it possible to collect a substantial amount of data from various sources of COBOL programs. These data will provide the opportunity for more extensive and critical analysis of the Software Science metrics, and their applicability to such important areas as programming time prediction and error prediction.

REFERENCES

[Halstead 77] Halstead, M.H., Elements of Software Science, North Holland, N.Y., 1977.

[Halstead 79] Halstead, M.H., "Advances in Software Science", in Advances in Computers, M.C. Yovits, Academic Press, New York, 1979.

[Jackson 75] Jackson, M., Principles of Program Design, Academic Press, 1975.

[Shen and Dunsmore 80] Shen, V., and Dunsmore, H., "A Software Science Analysis of COBOL Programs", Technical Report CSD-TR-348, Dept. of Computer Sciences, Purdue University, August 1980.

[Zweben and Fung 79] Zweben, S.H. and Fung, K.C., "Exploring Software Science Relations in COBOL and APL", Proceedings of COMPSAC 79, Chicago, Ill., Nov. 1979, 702-707.

APPENDIX-A

DESIGN DOCUMENT

This appendix provides the explicit description of each individual module in the analyzer program. Included for each module is a detailed functional description (with a list of subprocesses) and its interfaces with the other modules in the program.

Main Module: CNTPGM

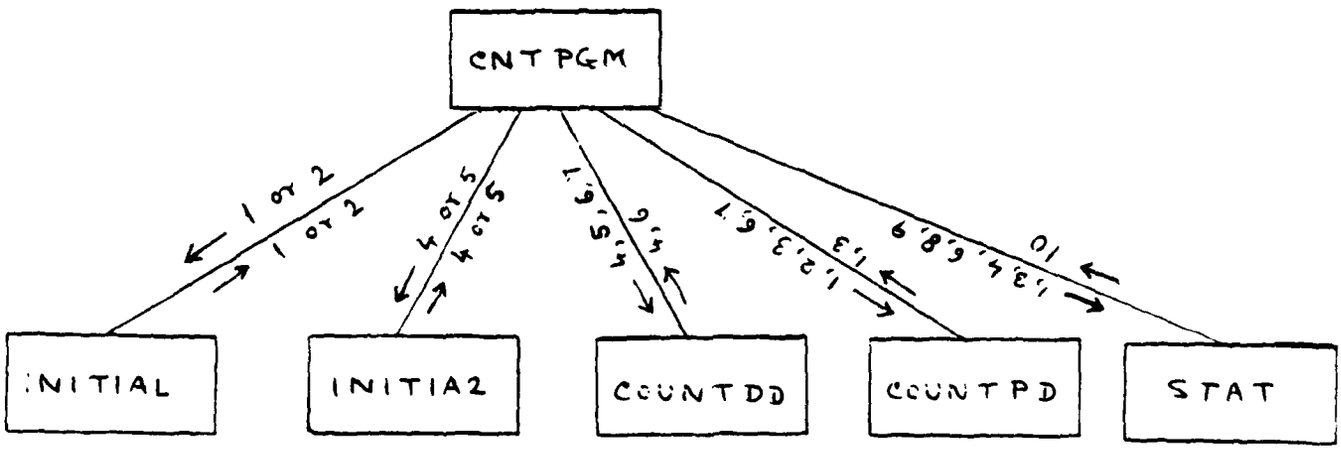
Functional Description:

This is the control module of the analyzer.

Subprocesses:

1. Initialize history list, namely six different tree roots.
2. Define the counting strategy for both data and procedure divisions (INITIA2, INITIAL).
3. Get author-ID and program-ID if the program is to be run for data collection.
4. Count data division (COUNTDD).
5. Count procedure division (COUNTPD).
6. Compute software metrics and print a report (STAT).

Interface:



OUT IN

1. 1. TOP-1 : Top of the operator tree for Procedure division (PD)

2. 2. TOP-2 : Top of the noise tree for PD.

3. 3. TOP-3 : Top of the operand tree for PD.

4. 4. TOP-6 : Top of the operator tree for data division (DD).

5. 5. TOP-7 : Top of the noise tree for DD.

6. 6. TOP-8 : Top of the operand tree for DD.

10. Final report of the analysis.

7. CIRPTR : Pointer to last token of the circular list.

8. LAB-ID : 2-digit Lab-ID.

9. AUTH-ID : 4-digit author-ID.

Module : INITIAL

Functional description:

Keywords and action pairs in the operator (OPER) and Noise (NOISE) files define the counting strategy of the Procedure division. This subprogram translates the counting strategy into operator and noiseword trees which will be used in the subsequent analysis.

Subprocesses:

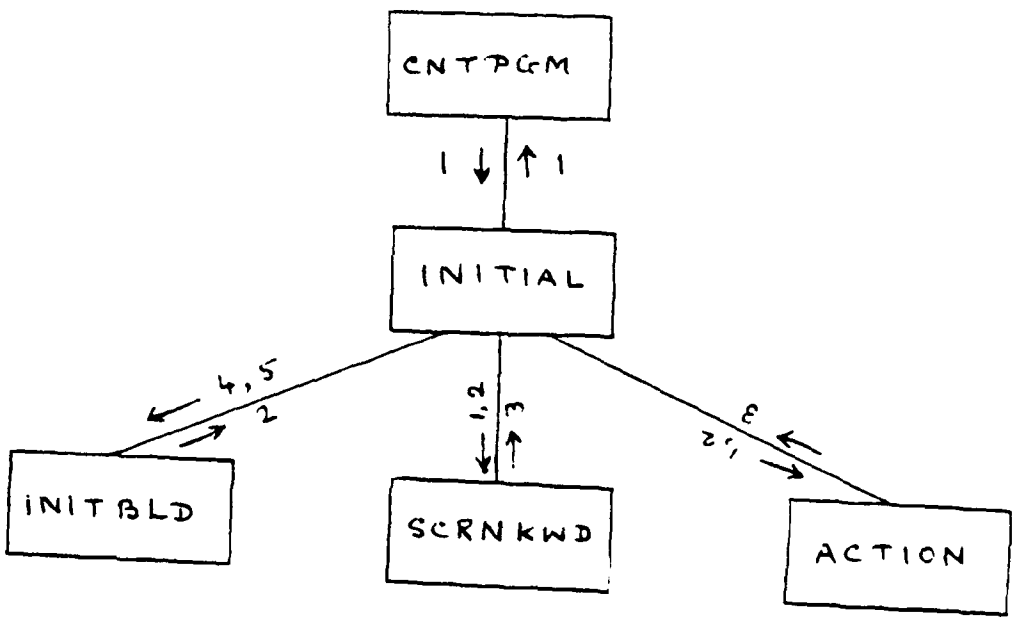
1. Capture a keyword from the input record of OPER file and put into a list structure (INITBLD).

2. Examine the operator tree and determine if the keyword has been linked into the tree. If it is not already present, the new keyword is inserted into the tree (SCRNKWD).

3. Process each action pair after the keyword. This process involves capturing the relationship symbol and the associated keyword, and constructing the associate list that implements the relationships between two keywords (ACTION).

4. Repeat the above steps with NOISE file.

Interface:



OUT IN

1. 1. TOP : Root of a given tree (Operator or Noiseword tree for PD).

2. 2. TRIAL-PTR : Address of the linked-list structure of the captured keyword/noiseword

3. SCRNKWD : Returned tree node of the screened keyword/noiseword

4. NEXT-CARD : Current input line image

5. CURSOR : Pointer to the current character in NEXT-CARD

Module: INITIA2

Functional Description:

This subprogram constructs the fundamental data structure needed to process the data division of a COBOL program. In particular, it translates the appropriate counting strategy into operator and noise trees for analyzing the data division.

INITIA2 has exactly similar structural and functional organization as INITIAL with the exception that INITIA2 uses the operator and Noise files for data division whereas INITIAL uses the operator and Noise files for Procedure divisions [see Appendix-B]. The detail module decription for INITIA2 is omitted in order to avoid repetition.

Module : INITBLD

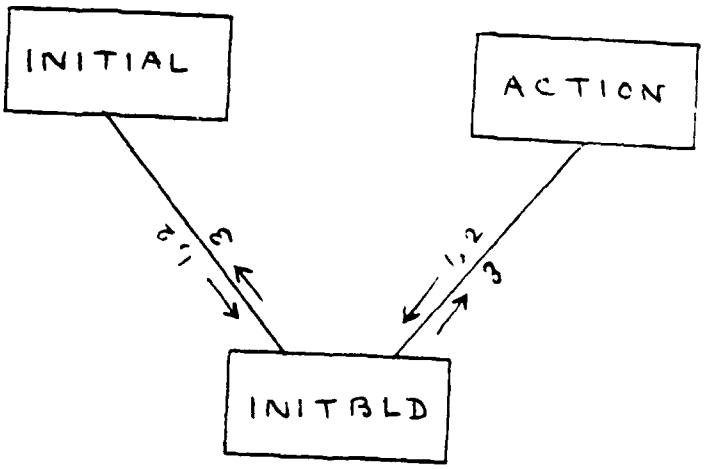
Functional Description:

This subprogram picks up a token from the input instruction record and constructs a linked list structure for that token.

Subprocess:

1. Assemble a token string by collecting one character at a time until a blank at end of card is reached.
2. Transform the token string into a linked list structure.
3. Return the address of the linked list structure.

Interface:



OUT

1. NEXT-CARD : Current input line image
2. CURSOR : Pointer to the current character in next-card.
3. TRIAL-PTR : Address of the linked list structure of the captured keyword/noiseword.

Module: SCRNKWD

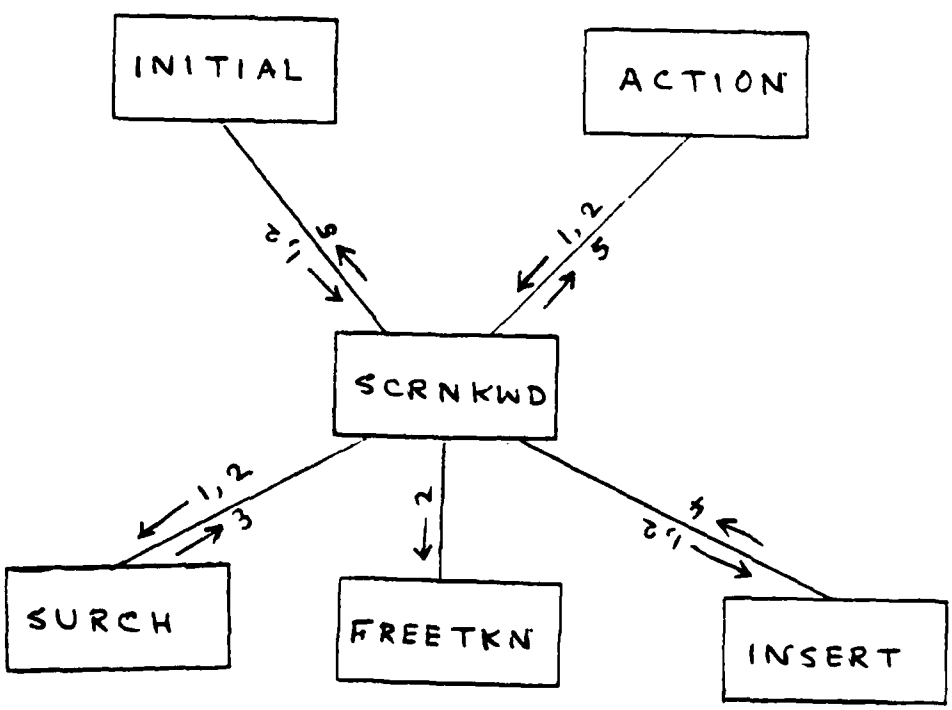
Functional Description:

This subprogram determines if the given keyword/noiseword captured by INITBLD is a new member or not. If it is an old member, the storage of the given keyword/noiseword is freed; otherwise it is inserted into the tree.

Subprocesses:

1. Traverse the appropriate tree to determine if a match in the tree can be found for the given keyword/noiseword (SURCH).
2. If a match exists, the storage of the keyword/noiseword is freed (FREETKN).
3. If a match cannot be found, the new keyword/noiseword is linked into the tree (INSERT).

Interface:



OUT IN

1. 1. TOP : Root of a given tree

2. 2. TRIAL-PTR : Address to the linked list structure of the keyword/noiseword.

3. SURCH : returned tree node of the matched keyword/ noiseword or "null".

4. SAVE : tree node of the insertion

5. SCRNKWD : returned tree node of the screened keyword/noiseword.

Module: ACTION

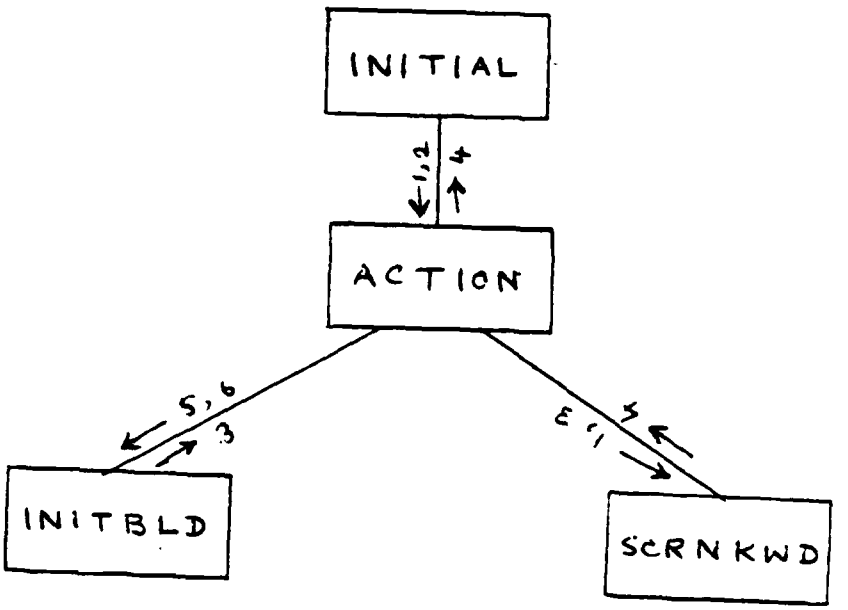
Functional Description:

This subprogram captures the relationship symbol (e.g. >>,=, <) and the associated keyword of the action pairs. It also contructs the associate-list that implements the relationship between the keywords.

Subprocesses:

1. Capture the first character of the relationship symbol.
2. If the first character is a `=', process the next keyword as a synonym through the synonym list.
3. If the first character is a `>', process the next keyword as a context sensitive element through the associate-list.
4. If the first character is a `<', process the next keyword as a context sensitive element through the associate-list.

Interface:



OUT	IN
1.	1. TOP: Root of the operator/noise tree.
	2. TRIAL-PTR: Address of first keyword in the list structure.
3.	3. INITBLD: Returned address to list structure of the associated keyword.

4. SCRNKWD: Returned treenode of the screened keyword/noiseword.

5. NEXT-CARD: Current input line image

6. CURSOR: Pointer to the current character in NEXT-CARD

Module: SURCH

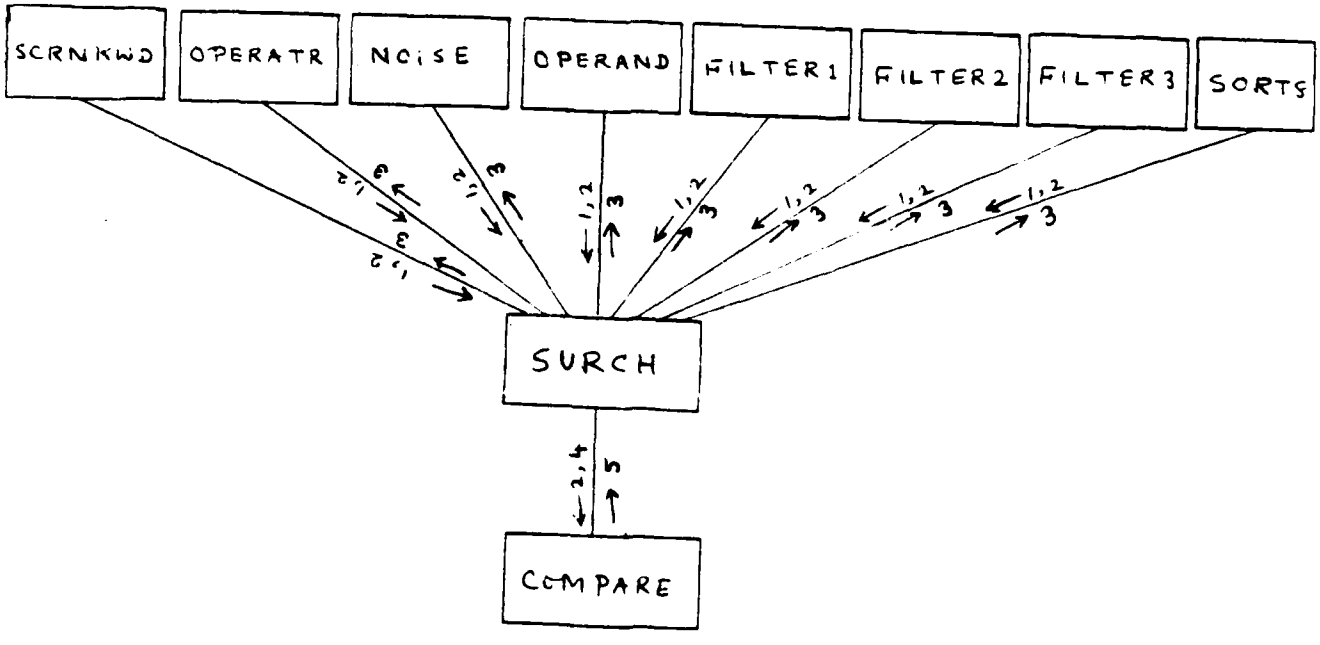
Functional Description:

This subprogram searches a binary tree by comparing the token character string associated with each node of the tree until a match is found or end of search is encountered.

Subprocesses:

1. Compare the given character string with the string associated with the tree nodes.
2. If both the strings are equal, return the address of the tree node.
3. If the given string is larger, access the tree node on the left.
4. If the given string is smaller, access the node on the right.
5. Repeat the above steps until a match or end of search is encountered.

Interface:



OUT IN

1. TOP: Root of a given tree

2. 2. TRIAL-PTR: Address of the given token list.

5. COMPARE: Returned the result of comparison.

3. SURCH: Returned address of the tree node that matches or trial-pointer.

4. HELP: The address of the tree node to be compared.

Module: INSERT

Functional Description:

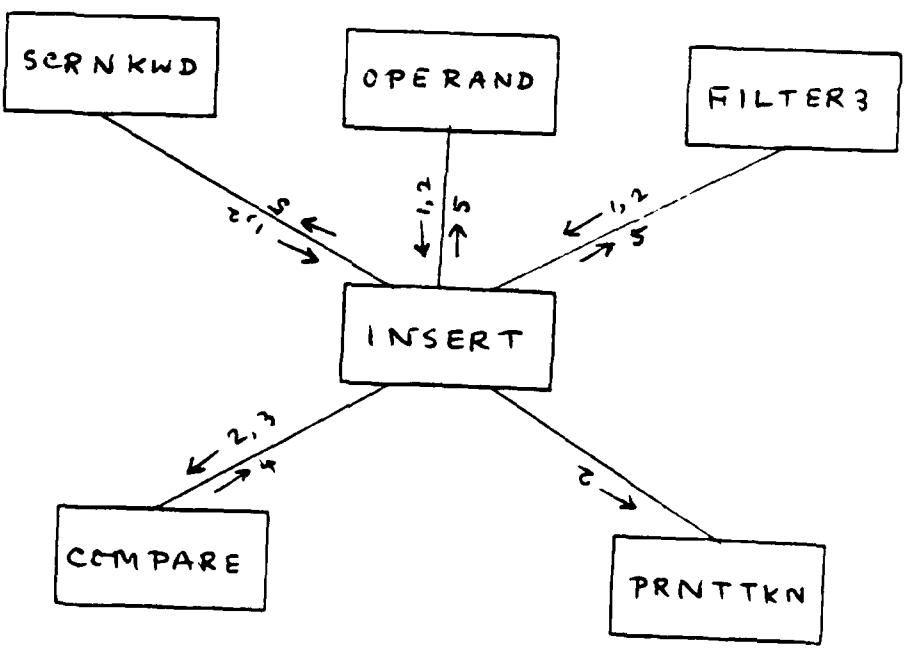
This subprogram inserts a new token into the given tree.

Subprocesses:

1. Compare the given token string with the character string associated with the tree node (COMPARE).

2. Insert the new token in the tree.

Interface:



OUT IN

1. TOP: Root of a given tree.

2. 2. TOKEN: Pointer to the current token list.

4. COMPARE: Returned result of comparison.

3. HELP: Pointer to the tree node to be compared.

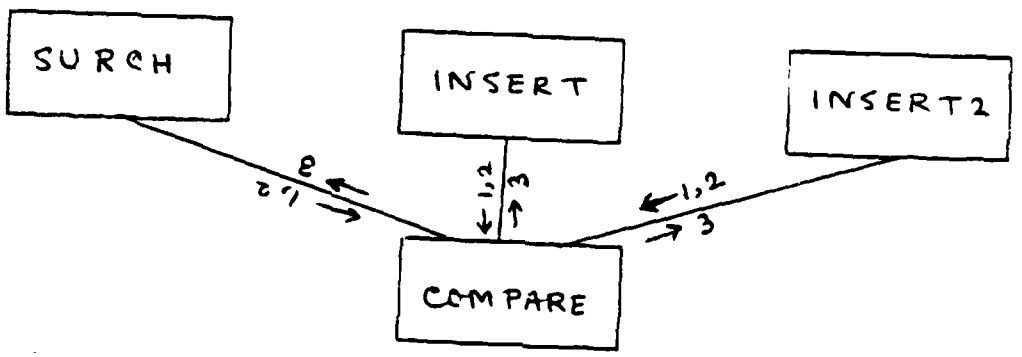
5. SAVE: Tree node address of insertion

Module: COMPARE

Functional Description:

This subprogram compares the character string contained in two given token list structures. It returns `GT' if the first character string is lexically of higher order than the second one. It returns `EQ' if they are equal and returns `LT' if the first string is of lower order than the second one.

Interface:



OUT IN

1. PTR-1: Pointer address to first character string

2. PTR-2: Pointer address to second character string.

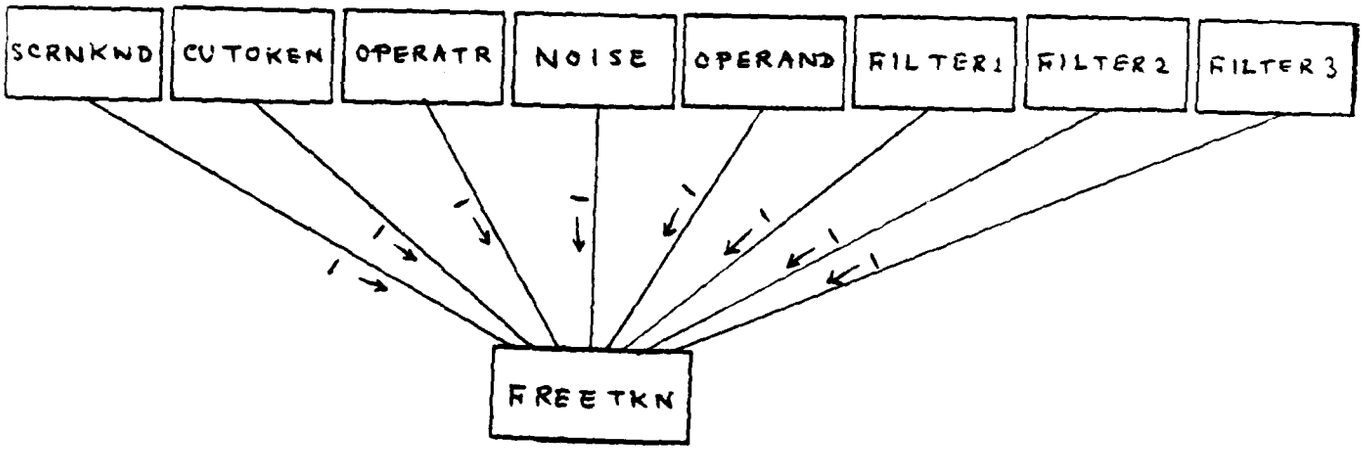
3. COMPARE: returned result of comparison.

Module: FREETKN

Functional Description:

This subprogram frees the storage occupied by the token list when the pointer to that particular list is known.

Interface:



IN

1. START: Pointer to the token list to be freed.

Module: COUNTDD

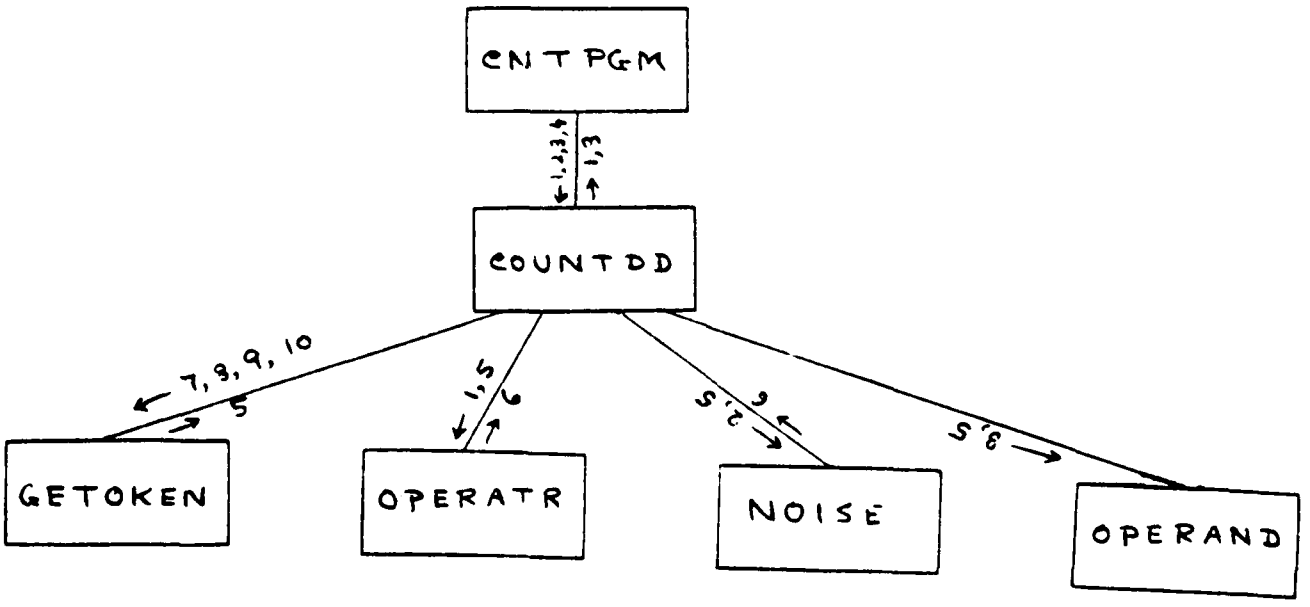
Functional Description:

This module scans the Data division of a COBOL program. Each occurrence of a token is classified on the basis of the counting strategy defined in INITIA2, and the frequency counts of each unique operator and operand is updated.

Subprocesses:

1. Capture a token from the input string (GETOKEN)
2. The current token is compared with the elements of the operator tree. If a match is found, then the current token is processed as an operator (OPERATR). Otherwise the token is passed to step3.
3. The incoming token is compared with the entries of the noise tree. If an identical token exists in this tree, the current token is processed as a noiseword (NOISE); otherwise the token is passed into step4.
4. In this stage, the current token is treated as an operand and processes accordingly (OPERAND).
5. Step1 thru step4 are repeated until the end of the Data division.

Interface:



OUT	IN
1.	1. TOP-6 : Root of the Operator tree for DD
2.	2. TOP-7 : Root of the Noise tree for DD
3.	3. TOP-8 : Root of the Operand tree for DD
	4. CIRPTR : Pointer to the entry of history list
5.	5. TOKEN : Pointer to the current token list
	6. RETURN-FLAG : Flag returned by OPERATR or NOISE

7. NEXT-CARD : Current input line image

8. CURSOR : Index to current character in NEXT-CARD

9. BUFF-CHAR : Buffer for the current token string in process

10. BUFF-PTR : Index to current character in buffer

Module: GETOKEN

Functional Description:

1. Find the first nonblank character of a token.

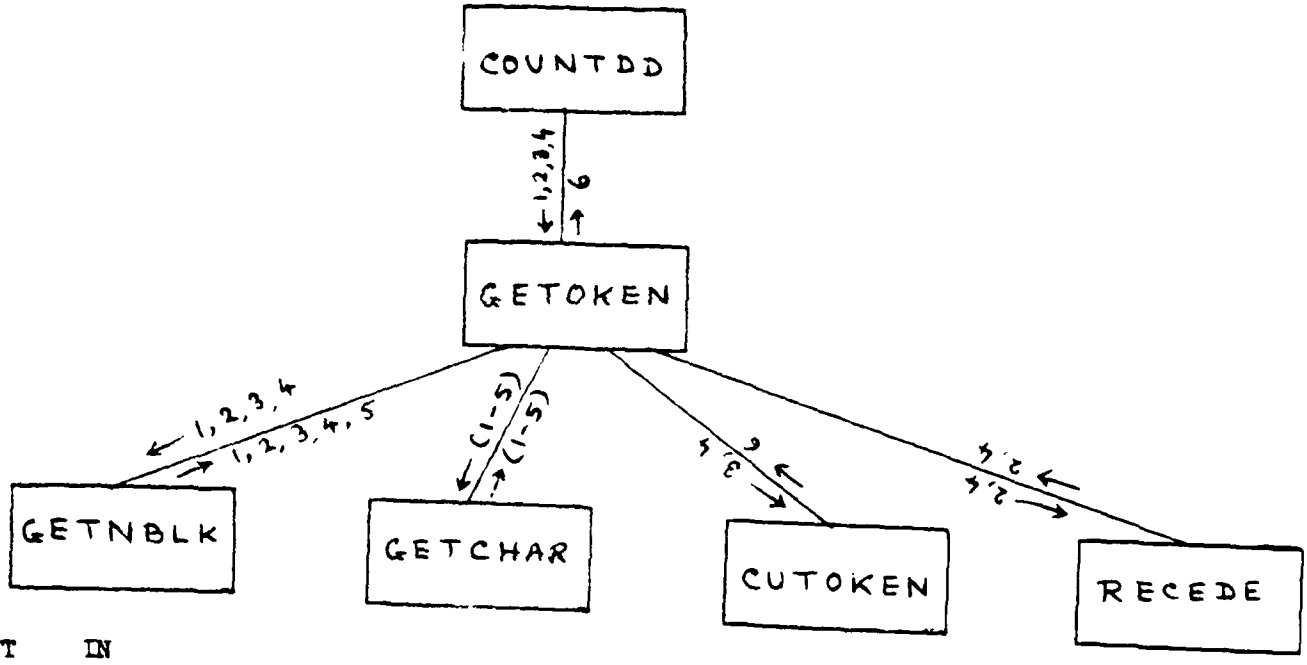
2. Capture all the adjacent characters starting with the first character until a blank is encountered.

3. If the last character of the string collected in step2 is a period, then ignore that period.

4. Generate a linked list structure for this new token.

5. Repeat step1 through step4 for the entire Data division.

Interface:



OUT IN

1. 1. NEXT-CARD : Current input line image

2. 2. CURSOR : Pointer to current character in NEXT-CARD.

3. 3. BUFF-CHAR : Buffer for current token string.

4. 4. BUFF-PTR : Pointer to current character in buffer.

5. 5. KAR : Current character in process.

6. 6. TOKEN : Pointer to token list.

Module: GETNBLK

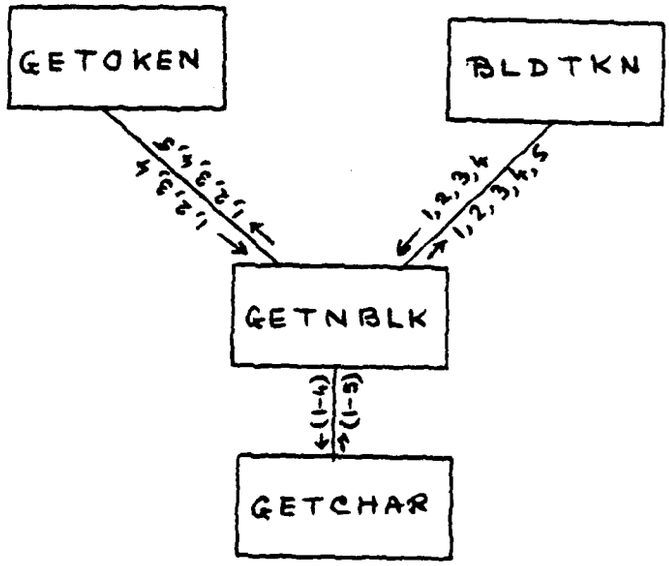
Functional Description:

This module finds the first nonblank character in an input record.

Subprocesses:

1. Check every character of the input string (starting from column 8 of the input record) until the first nonblank character is encountered.
2. Return the current nonblank character.

Interface:



OUT	IN
1.	1. NEXT-CARD : Current input line image
2.	2. CURSOR : Pointer to current character in NEXT-CARD.
3.	3. BUFF-CHAR : Buffer for current token string.
4.	4. BUFF-PTR : Pointer to current character in buffer.
5.	5. KAR : Current character in process.

Module: GETCHAR

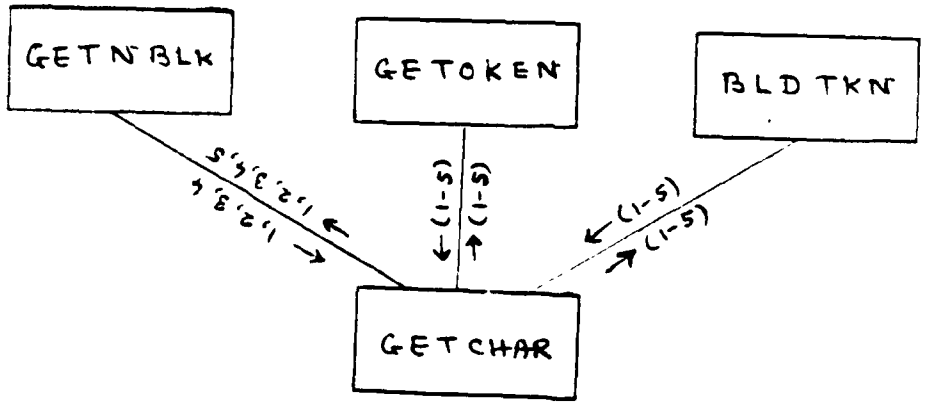
Functional Description:

This module returns the next relevant character in the input character stream. Characters in comments and labels, characters before column 8 and after column 72 are considered irrelevent.

Subprocesses:

1. Increment cursor by 1.
2. Read in another card when cursor is equal to 73.
3. Skip comments.
4. Skip labels.
5. Return the character to buffer and check for the overflow of the buffer.

Interface:



OUT	IN
1.	1. NEXT-CARD: Current input line image.
2.	2. CURSOR: Pointer to the current character in NEXT-CARD.
3.	3. BUFFER: Buffer for the current token string (may be partial).
4.	4. BUFF-PTR: Pointer to the last character of the token string in buffer.
5.	5. KAR: Current character in process.

Module: CUTOKEN

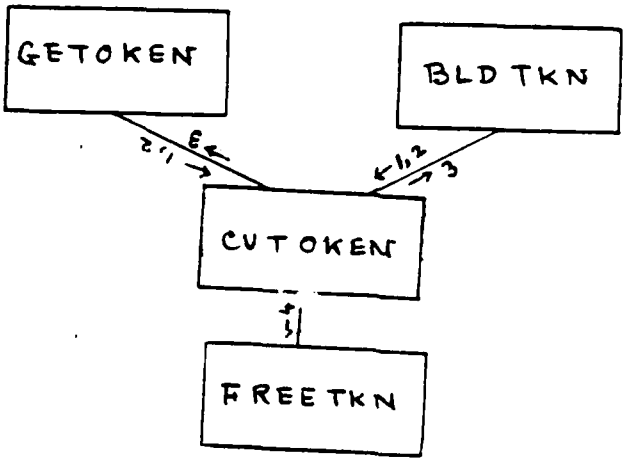
Functional Description:

This module constructs a linked-list structure of a particular token contained in the buffer area. Each node of the linked list contains only two adjacent characters of the token.

Subprocesses:

1. Allocate a node and initialize all fields in the node. Pointer to this node is returned to the calling module.
2. Fill up the node with first two characters of the buffer area.
3. Step1 and Step2 are repeated until all characters of the buffer are included in the list.
4. Empty the buffer area.

Interface:



OUT IN

1. BUFFER: Buffer for the currently captured token string.
2. BUFF-PTR: Pointer to the last character of the above token string.
3. TOKEN: Pointer to the token list constructed.
4. HELP: A temporary pointer which helps to hold the address of the last node in construction of the list.

Module: RECEDE

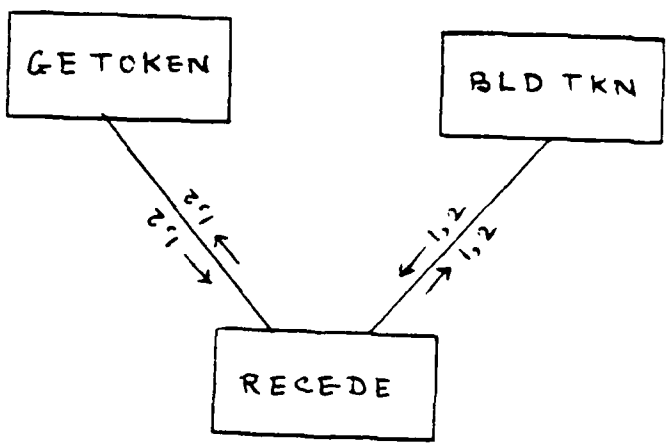
Functional Description:

In processing a character string, it becomes necessary to look ahead one or more characters. This module enables look ahead by providing a mechanism to recover the last character(s), if necessary, in the input string.

Subprocesses:

1. Decrement cursor by 1.
2. Decrement buffer pointer (BUFF-PTR) by 1.

Interfaces:



OUT IN

1. 1. CURSOR: Pointer to the current character in input string.
2. 2. BUFF-PTR: Pointer to the last character of the token string in buffer.

Module: OPERATR

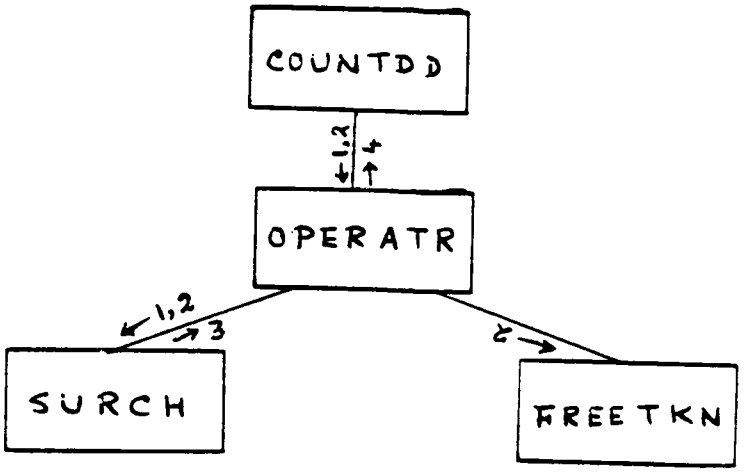
Functional Description:

This function subroutine returns "true" if a match of the current token exists in the operator tree; otherwise returns "false".

Subprocesses:

1. Search the operator tree and determine if the current token is an operator by comparing the current token with the members of the operator tree.
2. If the token is an operator, then increase the frequency counts of the token by 1 and free the incoming token list structure since only one version of the same token list is needed.
3..If the token is not an operator, return "false".

Interface:



OUT IN

1. 1. TOP-6 : Root of the operator tree for DD
2. 2. TOKEN : Pointer to token list in process
   3. SURCH : Returned tree node address of the matched token or "null".
4. RETURN-FLAG : Flag containing 1 or 0 depending on whether the current token exists in the operator tree or not.

Module: NOISE

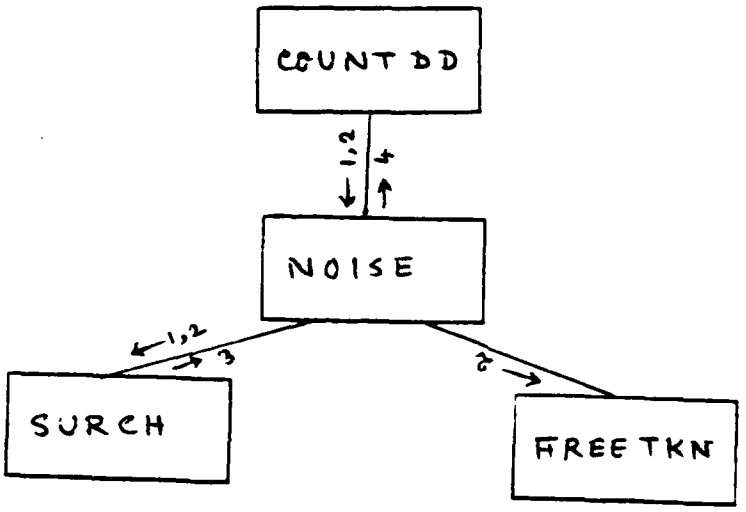
Functional Description:

This function subroutine returns "true" if the current token is found to be a noiseword. Otherwise it returns "false". Noisewords are ignored in the present analysis.

Subprocesses:

1. Search the noise tree to find whether the current token is a noiseword, by comparing the token with each member of the noise tree.
2. If the token is found to be a noiseword then return "true" and free the current token list; otherwise return "false".

Interface:



OUT IN

1. 1. TOP-7 : Root of the noise tree for DD
2. 2. TOKEN : Pointer to current token list
   3. SURCH : Returned tree node address of the matched token or "NULL"
4. RETURN-FLAG : Flag containing "true" or "false", returned by NOISE.

Module: OPERAND

Functional Description:

This module treats every incoming token as an operand. It searches the operand tree and inserts the incoming token into the tree only if it is a new member.

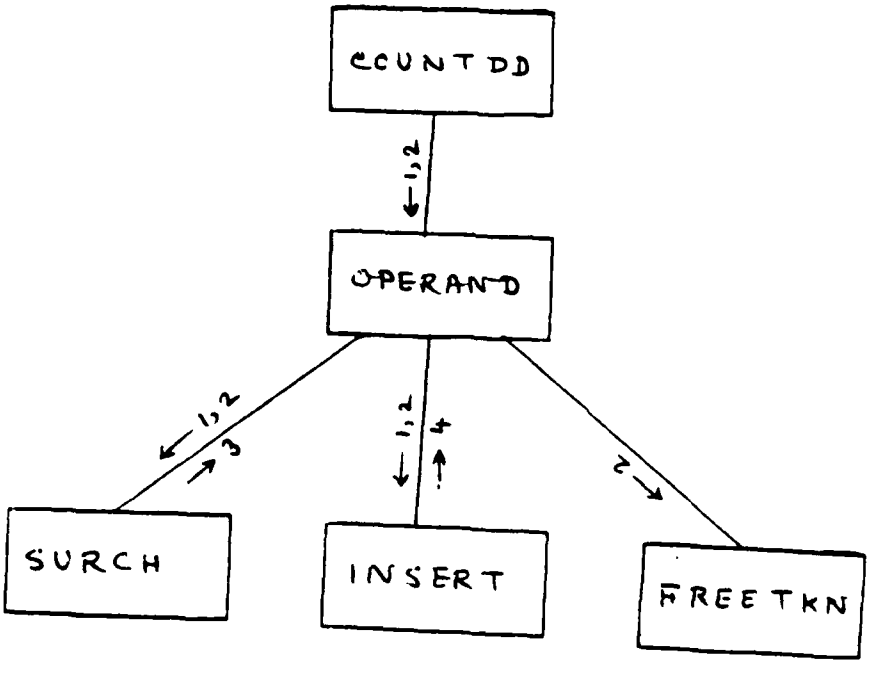
Subprocesses:

1. Search the operand tree and compare the incoming or current token with the members of the tree.

2. If the current token is found to be a new member, then insert this token into the operand tree and update the frequency count.

3. If the token already exists in the operand tree, then free the token list and increase the frequency count of the existing member by 1.

Interface:



OUT IN

1. 1. TOP-8 : Root of the operand tree for DD

2. 2. TOKEN : Pointer to current token list.

3. SURCH : Returned tree node address of the matchedf token or "NULL".

4. SAVE : Tree node address of insertion.

Module: COUNTPD

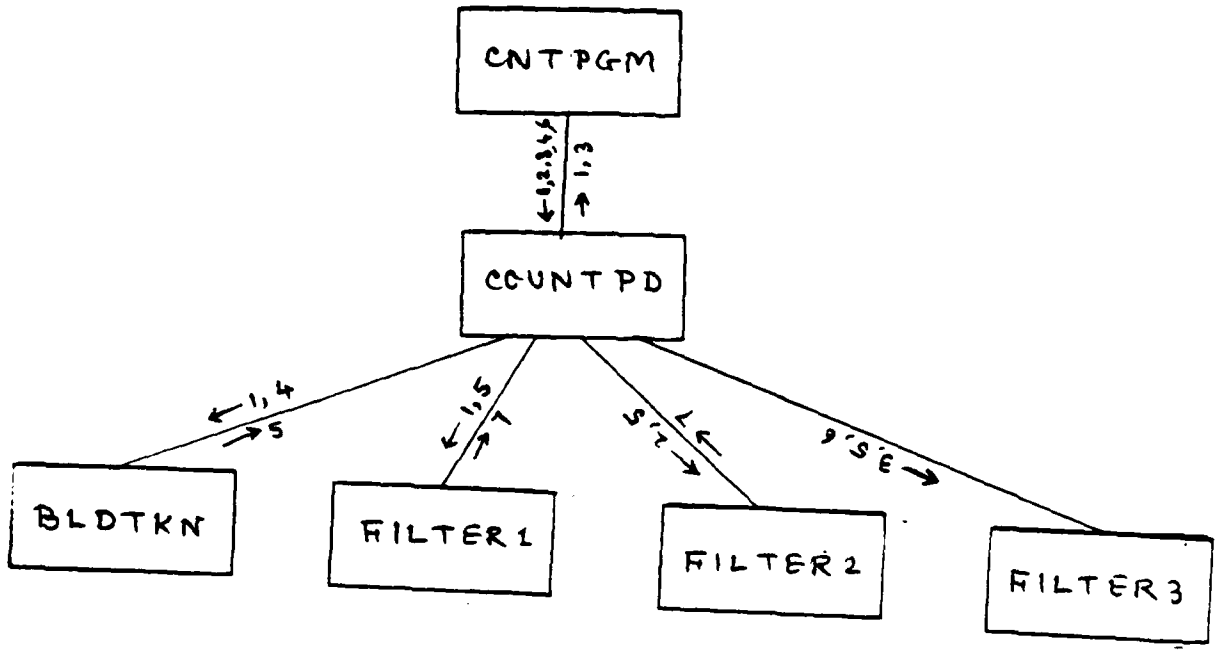
Functional Description:

It scans the procedure division of a COBOL program. Each occurrence of a token is classified according to a counting strategy defined in INITIAL and the frequency count of each unique operator, noiseword, or operand is updated.

Subprocesses:

1. Capture a token from the input (BLDTKN)
2. The token is compared with entries in the operator tree. If a match is found, the token is processed as an operator (FILTER1). Otherwise continue to the next step.
3. The token is compared with entries in the noise tree. If a match is found, the token is processed as a noiseword (FILTER2). Otherwise proceed to the next step.
4. The token is processed as operand (FILTER3).
5. Repeat the above process until end of program

Interface:



OUT	IN
1.	1. TOP-1 : Root of the operator tree for Procedure division
2.	2. TOP-2 : Root of the noiseword tree for Procedure division
3.	3. TOP-3 : Root of the operand tree for Procedure division
4.	4. CIRPTR : Pointer to entry of the history list
6.	6. TOP-8 : Root of the operand tree for DD
	7. DONE-FLAG : Flag returned by Filter 1 or Filter 2
5. TOKEN : Resultant pointer to the token list by BLDTKN	

Module: BLDTKN

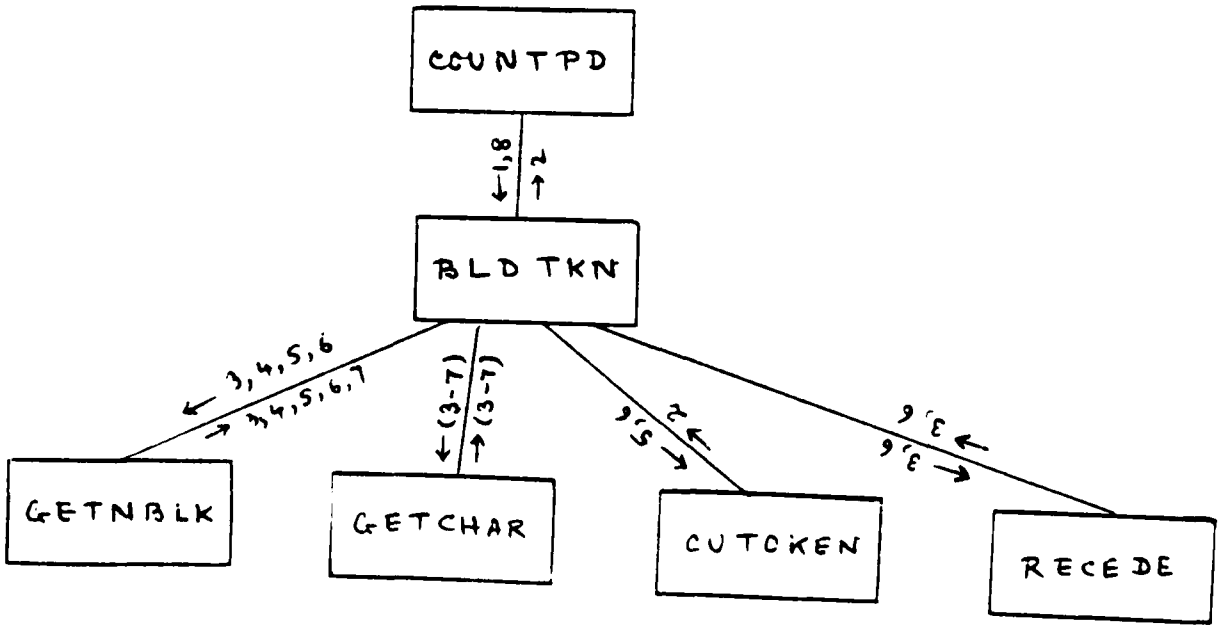
Functional Description:

It scans the source program input stream and captures the next token on the basis of a set of delimeters (e.g. blank, period, comma,*,+,-,/ etc.) defined in the COBOL language.

Subprocesses:

1. If the first nonblank character of a potential token string is a period, comma, or right parenthesis, then skip the character.
2. Capture `*` and `**`
3. Capture `/`,`+`,`-`,`>`,`<`,`=`,`(`
4. Capture literals enclosed by quotes (subroutine LITERAL)
5. Capture character strings delimited on the right by blank, `*`,`/`,`+`, `>`,`>`,`=`,`(`,`)`,`,`,`.`, and `-`. Embedding `.` or `-` is allowed.
6. Character strings of `CALL`,`PERFORM`,`ALTER`,`GO` and `NOTE` are processed separately by the subroutine named CALL-IT, PERFORM, ALTER, GOTO and NOTE respectively.

Interfaces:



OUT	IN
	1. TOP-1 : operator tree for new insertion of operator
2.	2. TOKEN : pointer to token list
3.	3. CURSOR : index to current character in NEXT-CARD
4.	4. NEXT-CARD : current input line image
5.	5. BUFFER : Buffer for current token string in process
6.	6. BUFFER-PTR : Index to current character in Buffer
7.	7. KAR : current character
	8. CIRPTR : Pointer to entry of the history list

Module: FILTER1

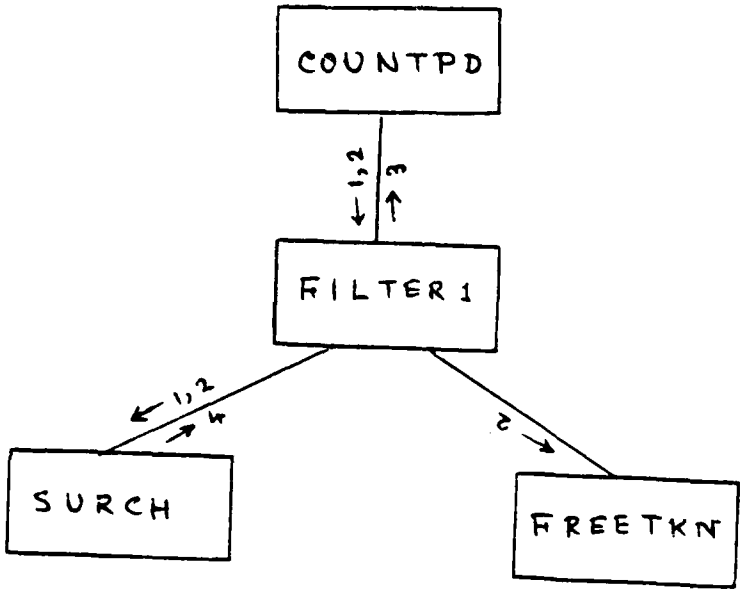
Functional Description:

This function subroutine returns "true" and updates the frequency count of a token captured by BLDTKN if a match is found in the operator tree. Otherwise it returns "false" for passage of the token to FILTER2.

Subprocesses:

1. Search the operator tree to determine if the token is an operator. If it is not an operator then exit this function and return "false". Otherwise process the token according to the following steps.
2. Increase the frequency count of this token by one, and free the incoming token list structure since only one version of the same token list is needed.
3. If the token found is potentially sensitive to past tokens, the history list containing the past ten operators should be examined. If indeed the token can affect a past token or be affected by a past token, appropriate action should be taken to address this situation.
4. Update the history list for the new token.
5. Return `true´ to COUNTPD.

Interface:



OUT IN

1. 1. TOP-1 : Root of the operator tree

2. 2. TOKEN : Pointer to token list in process

4. SURCH : Returned tree node address of the matched token or `null´ for non-match

3. FILTER1 : Returned `true´ or `false´ for `confirmed operator´ or `confirmed non-operator´.

Module: FILTER2

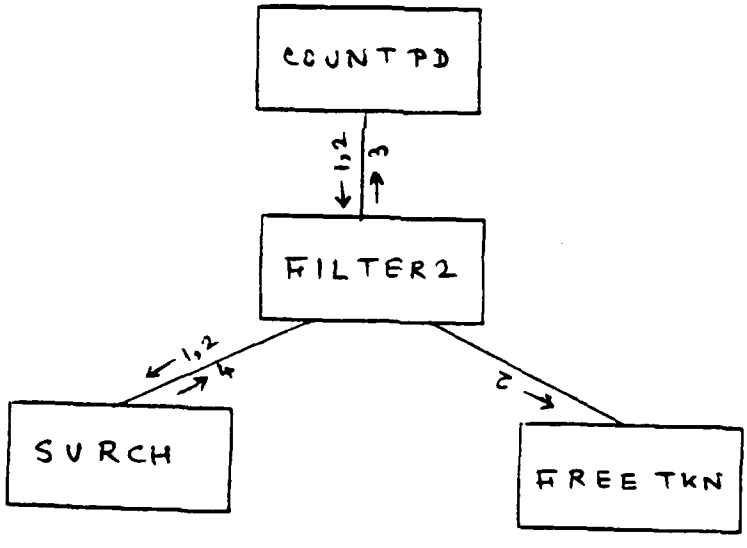
Functional Description:

This function subroutine returns "true" if a match is found in the noise tree. Since keeping counts of noisewords serves no purpose for the present analysis, noisewords are not counted.

Subprocess:

1. Search the noise tree to determine if current token is a noiseword. If it is then free the token and return "true". Otherwise return "false".

Interfaces:



OUT IN

1. 1. TOP-2 : Root of the noise tree

2. 2. TOKEN : Pointer to token list in process

4. SURCH : Returned tree node address of the matched token or `null' for non-match.

3. FILTER2 : Returned "true" or "false" for confirmed noiseword or confirmed nonnoiseword

Module: FILTER3

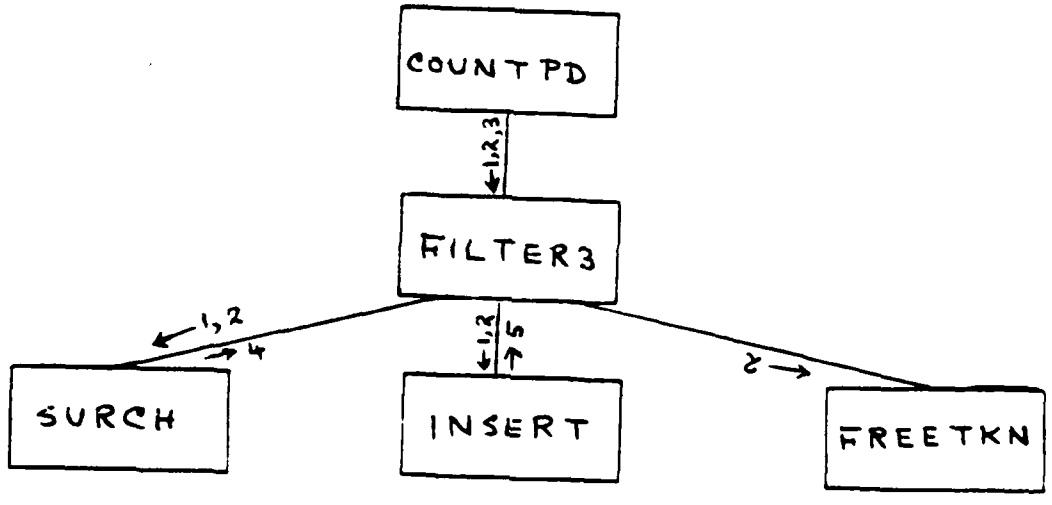
Functional Description:

It treats every incoming token as an operand. Most operands have been defined in the operand tree for the Data division. Therefore the DD operand tree is searched first to avoid duplicate representation of a unique token. The PD operand tree is searched next and only new operands are inserted into this tree.

Subprocesses:

1. Search the DD Operand tree to determine if the token has been declared in Data division. If it is, free the token list and use the one already in use.
2. Search the PD operand tree to determine if the token is already in the tree or not. If it is not, insert the token into the tree.
3. Increase the frequency count by 1.

Interface:



OUT IN

1. 1. TOP-3 : Root of the operand tree
2. 2. TOKEN : Pointer to token list
   3. TOP-8: Root of the operand tree for DD.
   4. SURCH: Returned tree node address or "null" for unmatched token.
   5. SAVE: Tree node address of insertion.

Module: STAT

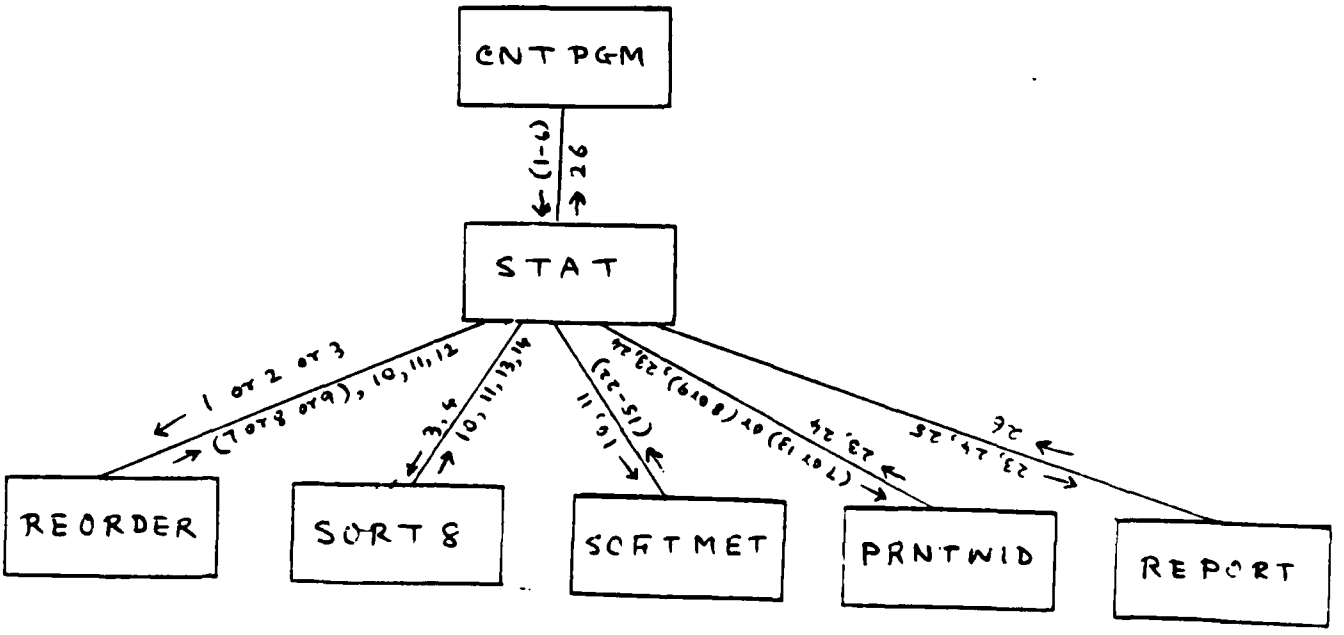
Functional Description:

This module produces readable printouts for all the parameters of software science.

Subprocesses:

1. Sort the operator and operand trees of both data and procedure divisions so that the tokens are arranged in order of frequency counts. Also calculate the number of unique operators/operands and the total number of operators/operands during the sort process (REORDER and SORT8)
2. Compute the values of all software science metrics (SOFTMET)
3. Generate seperate files containing the detailed information (e.g. frequency distribution of all tokens, summary record for each division) for data and procedure divisions, as well as for the whole program (PRNTWID)
4. Use the files generated in step 3 to print out the appropriate report (REPORT)

Interface:



OUT IN

1. 1. TOP-6: Root of DD operator tree.

2. 2. TOP-1: Root of PD operator tree.

3. 3. TOP-3: Root of PD operand tree.

4. 4. TOP-8: Root of DD operand tree.

5. Lab-ID: 2-digit Lab-ID (required only for student's job)

6. AUTHOR-ID: 4-digit author ID (required only for students' job)

7. 7. TOP-9: Root of the frequency tree of DD operator

8. 8. TOP-4: Root of the frequency tree of PD operator.

9. 9. TOP-5: Root of the frequency tree of PD operand

10. 10. UNIQUE: Number of unique operators/operand

11. 11. OCCURRENCE: total number of operators/operands.

12. 12. EOS: Number of statements in data/procedure division

13. 13. TOP-10: Root of the frequency tree of DD operand.

14. ETA2-INTERSECT: Common elements between DD and PD operand trees

15. N: Program length

16. ETA: Vocabulary

17. NH: Estimated program length

18. V: Program volume

19. LH: Program level

20. LAMBDAH: Language level

21. INTELL: Intelligence content

22. EFFORT: Programming effort

Software Science Metrics

23. 23. SYSUTO: File containing summary record and frequency counts of Data division.

24. 24. SYSUTI: File containing summary record and frequency counts of Procedure division.

26. 26. FINAL output of the Analyzer

25. SYSUT2 : File containing the summary record for the whole program

Module: REORDER

Functional Description:

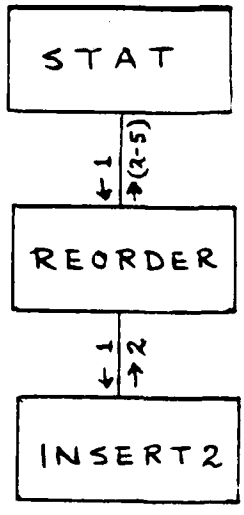
This module sorts (according to frequency counts) the operator tree and the operand tree of the procedure division as well as the operator tree of the data division. It also finds the number of unique operators (operands) and total number of operators (operands) while sorting the trees.

Subprocess:

1.Traverse the operator/operand trees using a postorder traversal algorithm

2. The visited node is pruned from the original tree and is inserted into the frequency tree, in which all the tokens are arranged in sequential order of frequency.

Interface:



OUT IN

1. 1. ROOT: Root of the given tree

2. 2. FREQ-TOP: Root of the frequency tree

3. UNIQUE : # of unique operators/operands

4. OCCURENCE: total # of operators/operands

5. EOS: # of statement in Data division/Procedure division

Module: INSERT2

Functional Description:

This module inserts the tree node carried by `root´ into the frequency tree topped by `Freq-top´. The insertion procedure of this routine differs from that of INSERT in the following ways.

(a) The sort key is the frequency count and then the character string of the token.

(b) The inserted unit is a tree node instead of token pointer.

(c) No storage allocation is needed. In other words, building the frequency tree does not require extra storage.

Subprocesses:

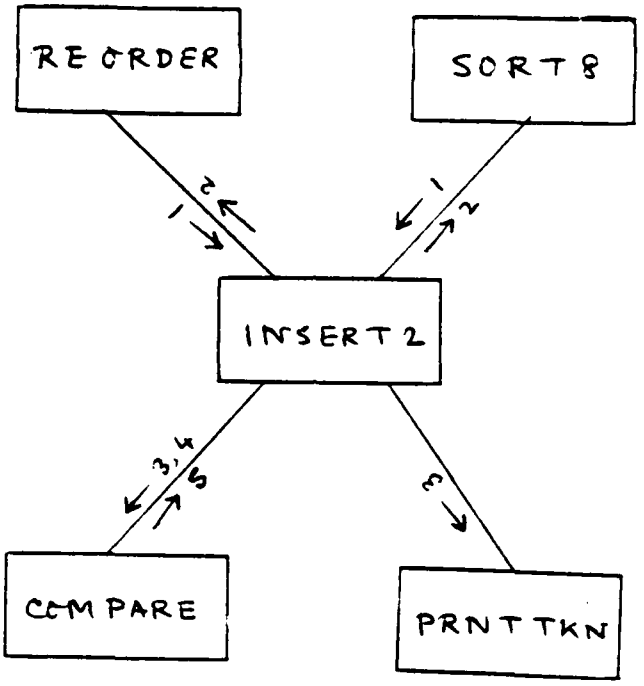
1. Use the root of the original tree as the top of the frequency tree.

2. If the frequency count of the next vistited node is greater than the frequency count of the frequency top, go to the right and insert the current node into the frequency tree if right pointer is null.

3. If the frequency count of the next visited node is less than that of the frequency top then go to the left node. If the left pointer is null then insert the current node into the frequency tree.

4. If the frequency count of the next visited node is equal to the frequency count of the frequency top, then insert the current node depending on the lexical order of the token strings.

Interface:



OUT IN

1. ROOT: Root of a given tree.
5. COMPARE: Returned the result of comparison between the token strings.

2. FREQ-TOP : Root of the frequency tree
3. HELP1 : Pointer to character string 1
4. HELP2 : Pointer to character string 2

Module: SORT8

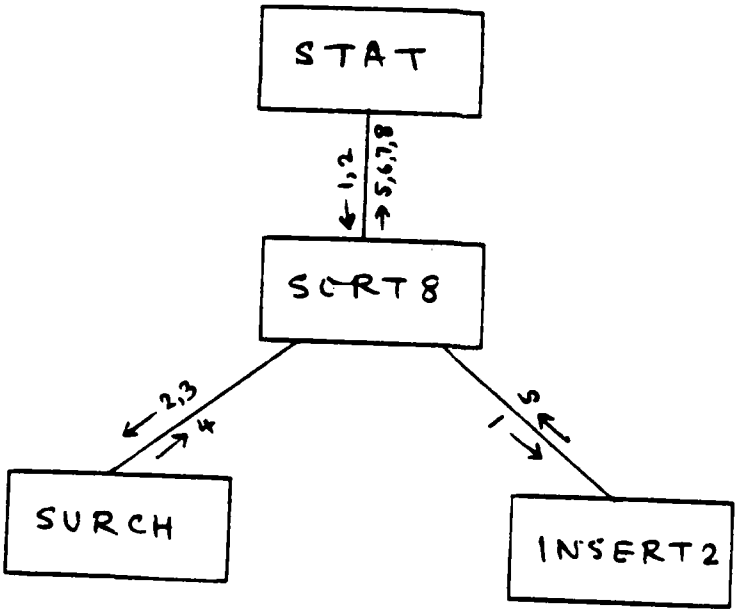
Functional Description:

This module sorts the operand tree of the Data division. It also finds the total number of common operands between the Data and Procedure divisions.

Subprocesses:

1. For each token in the DD operand tree, search the operand tree of the procedure division to find if the current token of the DD operand tree also exists in the operand tree of the Procedure division.
2. If the current DD operand also exists in the operand tree of the Procedure division, then the number of common tokens is incremented by 1.
3. Step1 and Step2 are carried out for all the tokens in the DD operand tree.
4. Sort the DD operand tree using the same procedure as the REORDER routine.

Interface:



OUT IN

1. 1. TOP-8 : Root of the operand tree of DD

2. 2. TOP-3 : Root of the operand tree of PD

4. SURCH : Returned address of the matched tree node or `null'

5. 5. TOP-10 : Root of the frequency tree of DD operand

3. TKNPTR : Pointer to current token in DD operand tree

6. ET.2-INTERSECT : Common operands between DD and PD operand trees.

7. UNIQUE : Number of unique operators/operands.

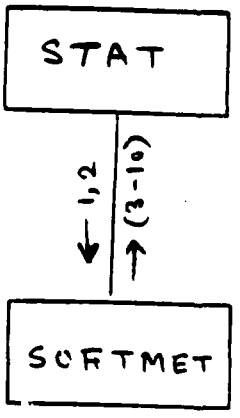
8. OCCURENCE : Total number of operators/operands.

Module: SOFTMET

Functional Description:

This routine produces the final values of all the software science metrics for Data and Procedure divisions.

Interface:



OUT IN

1. UNIQUE: Number of unique operators/operands for Data or Procedure division.
2. OCCURRENCE: Total number of operators/operands for Data or Procedure division.
3. N: Program length
4. ETA: Vocabulary
5. NH: Estimated program length
6. V: Program volume
7. LH: Program level
8. LAMDAH: Language level
9. INTELL: Intelligence content
10 EFFORT: Programming effort

Software Science Metrics

Module: PRNTWID

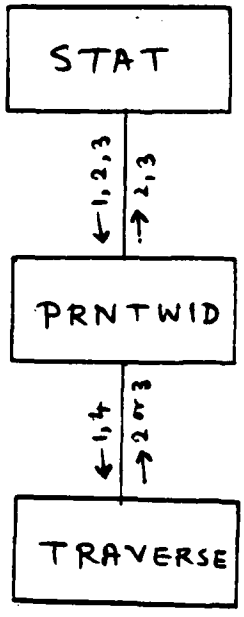
Functional Description:

This module generates files containing the frequency distribution of all the tokens in both Data and Procedure divisions.

Subprocesses:

1. Traverse the operator/operand tree using an inorder traversal algorithm.
2. Construct records containing the frequency distribution of the tokens

Interface:



OUT	IN
1.	1. TOP: Root of a given frequency tree
2.	2. SYSUT0 : File containing frequency distributions of tokens in Data division.
3.	3. SYSUT1 : File containing frequency distributions of tokens in Procedure division.

4. RW-REC: 80-character record of SYSUT0 or SYSUT1.

Module: TRAVERSE

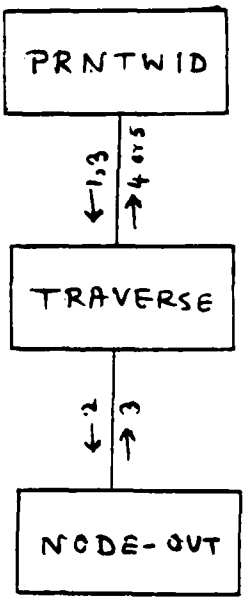
Functional Description:

This recursive subroutine is used to traverse a particular frequency tree and produces records of the file SYSUTO or SYSUT1

Subprocess:

1. At every node of the frequency tree, find the frequency counts of he token and write 80-character record (NODE- OUT).

Interface:



OUT IN

1. TOP : Root of a given frequency tree

3. PW-REC : 80 character record of SYSUTO or SYSUT1

2. TREE-NODE : Node unit of the operator/operand tree

4. SYSUTO : File containing information for Data division

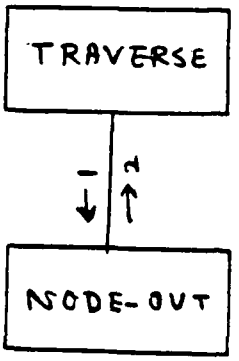
5. SYSUT1 : File containing information for Procedure division

Module: NODE-OUT

Functional Description:

Given a frequency tree node, this subroutine finds the frequency count of the token. It also writes 80-character record of SYSUT0/SYSUT1.

Interface:



OUT IN

1. TREE-NODE: Node unit of the operator/ operand tree.
2. PW-REC: 80-character record of SYSUT0 or SYSUT1

Module: REPORT

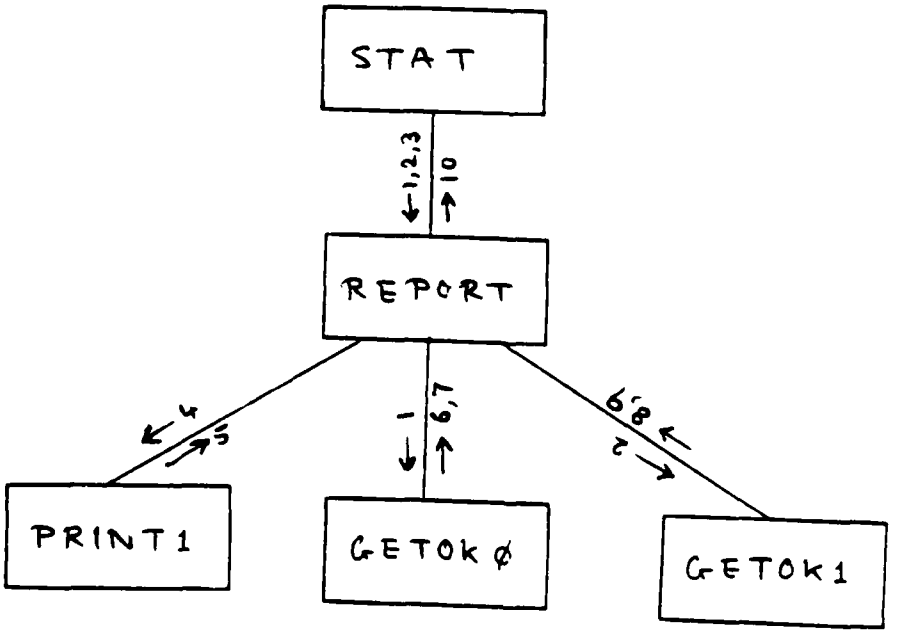
Functional Description:

This module produces well documented output, produced by the analyzer, for a COBOL program.

Subprocesses:

1. Read the files SYSUT0, SYSUT1 and SYSUT2.
2. Use SYSUT0 to produce the frequency distribution of the tokens as well as software metrics for the Data division.
2. Use SYSUT1 to produce the frequency distribution of the tokens as well as software metrics for the Procedure division.
3. Use SYSUT2 to provide the software metrics for the entire program
4. Print all the information of steps 2, 3, & 4.

Interface:



OUT IN

1. 1. SYSUT0 : All the information for DD

2. 2. SYSUT1 : All the information for PD

3. SYSUT2 : Summary record for the program

5. WIDIPLA : Output line of 133 characters

6. BUFFER0 : Buffer containing token strings of DD

7. FREQ0 : Frequency counts of tokens in DD

8. BUFFER1 : Token strings of PD

9. FREQ 1 : Frequency counts of tokens in PD

4. BODY : Information to be printed.

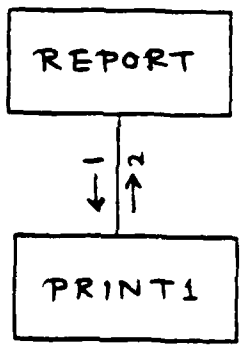
10. Final Output of the Analysis.

Module: Print1

Functional description:

This routine prints each output line of the final report.

Interface:



OUT IN

1. BODY: Information to be printed.

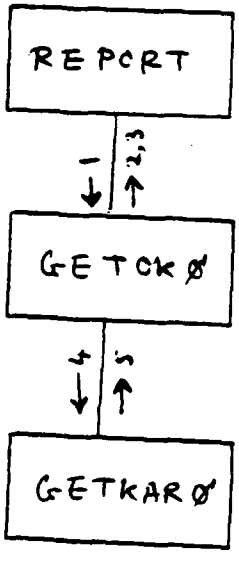
2. WIDIPLA: Output line of 133 characters.

Module: GETOKO

Functional Description:

It reads the file named SYSUTO, transfers characters from the input string into the buffer and also captures the frequency counts of the corresponding token string.

Interface:



OUT IN

1. SYSUTO: File containing the information of DD.

5. KAR: Current character in process.

2. BUFFERO: Buffer containing the token string of DD.

3. FREQO: Frequency counts of the token.

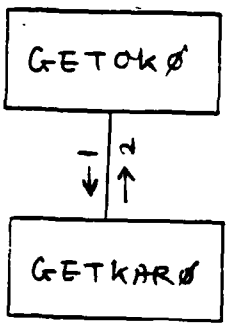
4. INPUT-RECO: Each record of SYSUTO.

Module: GETKAR0

Functional Description:

Given each record of the file SYSUT0, this routine captures one character at a time until the end of record is encountered.

Interface:



OUT IN

1. INPUT-REC0: Each 80-character record of SYSUT0.

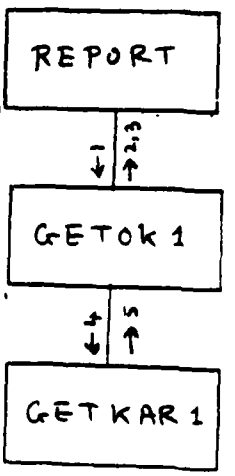
2. KAR: Current character in process.

Module: GETOK1

Functional Description:

It reads the file named SYSUT1, transfers characters from the input string into the buffer and also captures the frequency count of the associated token string.

Interface:



OUT IN

1. SYSUT1: File containing the information of PD.

5. KAR: Current character in process.

2. BUFFER1: Buffer containing the token string of PD.

3. FREQ1: Frequency count of the token.

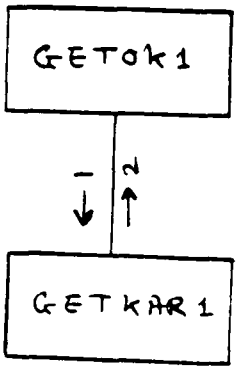
4. INPUT-REC1: Each record of SYSUT1.

Module: GETKAR1

Functional Description:

Given each record of the file SYSUT1, this routine captures one character at a time until the end of record is encountered.

Interface:



OUT IN

1. INPUT-REC1: Each 80-character record of SYSUT1.

2. KAR: Current character in process.

APPENDIX - B

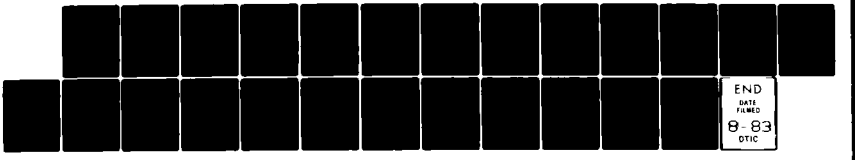
FILE DESCRIPTIONS

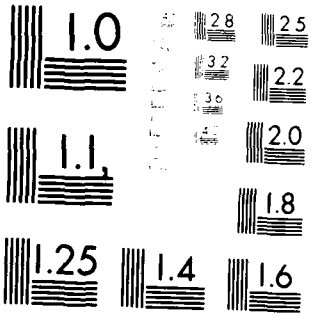
This appendix contains the operator and noiseword file listings used when the analyzer is run in default mode. Files for both Data and Procedure divisions are included. The members of these operator and noiseword files (B.1 to B.4) are used to construct and initialize the fundamental data structures, namely the operator and noise trees, used in analyzing a COBOL program. Finally, the production JCLs to run the analyzer are shown.

B.1 : DEFAULT OPERATOR FILE FOR DATA DIVISION (OPERO)

```
1.  ..
2.  FD
3.  LABEL
4.  BLOCK
5.  OMITTED
6.  DATA
7.  STANDARD
8.  VALUE
9.  LINAGE
10. FOOTING
11. TOP
12. BOTTOM
13. CODE-SET
14. RECORD
15. RECORDING
16. REDEFINES
17. PICTURE = PIC
18. USAGE
19. SIGN
20. LEADING
21. SEPARATE
23. OCCURS
24. TO
25. DEPENDING
26. ASCENDING
27. DESCENDING
28. INDEXED
29. SYNCHRONIZED = SYNC
30. JUSTIFIED = JUST
31. BLANK
32. RENAME
33. THRU
34. TIMES
35. DISPLAY
36. COMPUTATIONAL = COMP
37. COMPUTATIONAL-1 = COMP-1
38. COMPUTATIONAL-2 = COMP-2
39. COMPUTATIONAL-3 = COMP-3
40. LEFT
41. RIGHT
```

AD-A130 899 A SOFTWARE SCIENCE ANALYZER FOR COBOL REVISION(U) OHIO 2/2
STATE UNIV COLUMBUS COMPUTER AND INFORMATION SCIENCE
RES.. K C FUNG ET AL. 1982 OSU-CISRC-TR-83-2-REV
UNCLASSIFIED ARO-17150.3-EL DAAG29-80-K-0061 F/G 9/2 NL





B.2 : DEFAULT NOISEWORD FILE FOR DATA DIVISION (NOISE0)

1. ARE
2. IS
3. ON
4. BY
5. REPORT
6. LINKAGE
7. WORKING-STORAGE
8. FILE
9. CONTAINS
10. MODE
11. CHARACTERS
12. CHARACTER
13. LINES
14. WITH
15. AT
16. KEY
17. WHEN
18. DIVISION
19. SECTION

B.3 : DEFAULT OPERATOR FILE FOR PROCEDURE DIVISION (OPER1)

```
1.    *
2.    (
3.    +
4.    /
5.    .MOVE
6.    .DIVIDE
7.    .REWIND
8.    .CLOSE
9.    .SORT
10.   .EXIT
11.   .RELEASE
12.   .ACCEPT
13.   .SUBTRACT
14.   .COPY
15.   .SEARCH
16.   .GOBACK
17.   .RESET
18.   .READ
19.   .ELSE
20.   .UNSTRING
21.   .ADD
22.   .WRITE
23.   .CANCEL
24.   .STOP
25.   .COMPUTE
26.   .SET
27.   .DISPLAY
28.   .REWRITE
29.   .ENTER
30.   .RETURN
31.   .EXAMINE
32.   .READY
33.   .IF
34.   .OPEN
35.   .INSPECT
36.   .MULTIPLY
37.   AND
38.   WHEN
39.   FOR
40.   GIVING
41.   EXCEPTION
42.   NOT
43.   EXTEND
44.   FILE
45.   FIRST
46.   I-O
47.   POSITIONING
48.   REEL
49.   REPACE
50.   UP
51.   POINTER
52.   TALLYING
```

53. DELIMITED
54. DEPENDING
55. DISP
56. LEADING
57. BEFORE
58. ERROR
59. TIMES
60. END
61. TRANSFORM
62. UNTIL
63. INTO
64. INVALID
65. REMAINDER
66. BEGINNING
67. COUNT
68 EXHIBIT
69 NEXT
70. UNIT
71. UPON
72. LOCK
73. USE
74. USING
75. VARYING
76. OR
77. REMOVAL
78. MERGE
79. DOWN
80. SUPPRESS
81. ENDING
82. ENTRY
83. OUTPUT
84. AFTER
85. ALL
86. REPLACING
87. ROUNDED
88. SEQUENCE
89. START
90. STRING
91. INPUT
92. CHANGED
93. CHARACTERS
94. NAMED
95. BY < REPLACE < UP < DOWN < DELIMITED
96. PROCEDURE < LABEL
97. FROM > CHARACTERS
98. CORRESPONDING = CORR
99. TO < EQUAL < PROCEED
100. ERROR > SIZE > ON
101. ON < DEPENDING
102. ELSE = OTHERWISE
103. TO < PROCEED
104. ON < OUTPUT < INPUT < I-O
105. DESCENDING > ON
106. GREATER = >

```
107.  END-OF-PAGE=EOP
108.  EQUAL==
109.  PROCEED < TO
110.  LESS = <
111.  PROCEDURE > OUTPUT > INPUT
112.  ALL > SEARCH
113.  OVERFLOW > ON
114.  ASCENDING > ON
115.  THROUGH = THRU
```

Note that the minus sign is not included in this list since it is indistinguishable in form from a hyphen. The analyzer resolves this ambiguity by context. Other context sensitive operator information appears explcitly in the action at the end of the file.

B.4 : DEFAULT NOISEWORD FILE FOR PROCEDURE DIVISION (NOISE1)

```
1.   ;
2.   THAN
3.   )
4.   THEN
5.   ,
6.   ADVANCING
7.   AT
8.   LINES
9.   RUN
10.  IS
11.  SKIP1
12.  SKIP2
13.  SKIP3
14.  EJECT
```

B.5 : File, INITW, which generates output for production purposes (see WIDJET in B.9).

```
1.   MODE = `PRODUCTION´
2.   GO
3.   PERFORM
4.   CALL
5.   ALTER
6.   NOTE
7.   -
8.   PROCEDURE
9.   PROGRAM-ID
10.  DATA
11.  $JOB
12.  IDENTIFICATION
```

B.6 : File, INITR, which generates output for debugging purposes (see OSUCNTPM).

```
1.  MODE= `DEBUG´
2.  GO
3.  PERFORM
4.  CALL
5.  ALTER
6.  NOTE
7.  -
8.  PROCEDURE
9.  PROGRAM-ID
10. DATA
11. $JOB
12. IDENTIFICATION
```

Note: `PRODUCTION´ is the production mode, which generates the regular output (see section 3.3). On the other hand, `DEBUG´ is the debugging mode. Its output consists of a trace of input stream tokens helpful for debugging purposes. The output of this mode is different from that obtained in production mode.

B.7 : PRODUCTION JCL (OSUCNTPM)

```
1.  //      EXEC PGM=OSUCNTPM
2.  //STEPLIB  DD  DSN= TS0618.LOADLIB,DISP=SHR
3.  //OPER  DD  DSN=TS0618.CSLIB(OPER1),DISP=OLD,UNIT=USERDA
4.  //OPER2  DD  DSN=TS0618.CSLIB(OPER0),DISP=OLD,UNIT=USERDA
5.  //NOISE  DD  DSN=TS0618.CSLIB(NOISE1),DISP=OLD,UNIT=USERDA
6.  //NOISE2  DD  DSN=TS0618.CSLIB(NOISE0),DISP=OLD,UNIT=USERDA
7.  //INIT  DD  DSN=TS0618.CSLIB(INITR),DISP=OLD,UNIT=USERDA
8.  //SYSPRINT  DD  SYSOUT=A
9.  //WIDIPLA  DD  SYSOUT=A
10. //WIDSUM0  DD DUMMY
11. //WIDSUM1  DD DUMMY
12. //WIDSUM2  DD DUMMY
13. //WIDFREQ  DD DUMMY
14. //SYSUT0  DD  DSN=TS0618.CIS2212.SYSUT0,UNIT=USERDA,DISP=OLD
15. //SYSUT1  DD  DSN=TS0618.CIS212.SYSUT1,UNIT=USERDA,DISP=OLD
16. //SYSUT2  DD  DSN=TS0618.CIS212.SYSUT2,UNIT=USERDA,DISP=OLD
```

This JCL invokes the analyzer and generates hard copy of the analysis report, but does not store the analyzer output into a separate disk file for future use.

B.8 : JCL FOR INVOKING WIDJET (INTERNAL)

```
1.      //    EXEC    PGM=IEBGENER
2.      //SYSUT2 DD SYSOUT=(A,INTRDR),DCB=(RECFM=FB,LRECL=80,BLKSIZE=80)
3.      //SYSPRINT  DD  SYSOUT=A
4.      //SYSIN DD DUMMY
5.      //SYSUT1 DD DSN=TS0618.WIDJET,DISP=OLD,UNIT=USERDA
6.      //      DD DDNAME=PROGRAM
```

This JCL file called INTERNAL is used to invoke the production JCL required to run the analyzer through the on-line terminal. In particular, WIDJET or WYLBUR users invoke INTERNAL, which in turn invokes the JCL file called WIDJET through the internal reader.

B.9 : PRODUCTION JCL (WIDJET)

```
0.01    //COUNTPGM JOB 'FRB080,212313080','ANALYZER, C.
0.02    /*JOBPARM  V=D
0.03    //      EXEC PGM=OSUCNTPM
0.04    //STEPLIB    DD  DSN=TS0618.LOADLIB,DISP=SHR
0.05    //SYSPRINT    DD  SYSOUT=A
0.09    //WIDIPLA  DD  SYSOUT=A
0.14    //OPER  DD DSN=TS0618.CSLIB(OPER1),DISP=OLD,UNIT=USERDA
0.15    //OPER2 DD DSN=TS0618.CSLIB(OPER0),DISP=OLD,UNIT=USERDA
0.16    //NOISE  DD DSN=TS0618.CSLIB(NOISE1),DISP=OLD,UNIT=USERDA
0.17    //NOISE2 DD DSN=TS0618.CSLIB(NOISE0),DISP=OLD,UNIT=USERDA
0.18    //INIT    DD  DSN=TS0618.CSLIB(INITW),DISP=OLD,UNIT=USERDA
0.21    //WIDSUM0  DD  DUMMY
0.22    //WIDSUM1  DD  DUMMY
0.23    //WIDSUM2  DD  DUMMY
0.24    //WIDFREQ  DD  DSN=TS0618.CIS212.FREQ,UNIT=USERDA,DISP=(MOD,CATLG),
0.241   //    DCB=(RECFM=FB,LRECL=80,BLKSIZE=80,DSORG=PS),
0.242   //    SPACE=(TRK,(50,20))
0.25    //SYSUT0    DD  DSN=TS0618.CIS212.SYSUT0,UNIT=USERDA,DISP=OLD
0.26    //SYSUT1    DD  DSN=TS0618.CIS212.SYSUT1,UNIT=USERDA,DISP=OLD
0.27    //SYSUT2    DD  DSN=TS0618.CIS212.SYSUT2,UNIT=USERDA,DISP=OLD
0.28    //SOURCE  DD *
```

This JCL file called WIDJET is used to run the analyzer and to collect Software Science data from student COBOL programs at Ohio State University. One should note the file `FIDFREQ' and its parameters. In particular, DSN = TS0618.CIS212.FREQ gives the file name on the disk where all the data from the analyzer is to be collected, and the SPACE parameters provides sthe primary and secondary storage allocation for this disk file. In the present case, the primary storage allocation for TS0618.cis212. would be 50 tracks and the secondary allocation is 20 tracks (up to 16 extent).

```
SET EXEC NOLOG TERSE
ON ERROR
ON ATTN
SET VALUE S0 LAST
IF ( S0 EQ 0.000) EXEC 11.01
COMMENT ************************************************
COMMENT THE ACTIVE FILE MUST BE EMPTY TO RUN THE
COMMENT ANALYZER.   TRY AGAIN AFTER SAVING THE FILE IF
COMMENT NEEDED AND ALSO CLEARING THE ACTIVE FILE.
COMMENT ************************************************
EXEC 54
COMMENT
COMMENT WHICH COURSE IS THIS(212 OR 313)?
READ STRING S0
IF(S0 EQ '212') EXEC 12
IF(S0 EQ '313') EXEC 61
EXEC 11.01
CLEAR ACTIVE
SET ESCAPE %
COMMENT WHICH OF THE FOLLOWING LAB NUMBERS DO YOU WANT ANALYZED?
COMMENT
COMMENT       02 03 04 05 06
COMMENT
READ STRING S1
IF(S1 EQ '02') EXEC 28
IF(S1 EQ '03') EXEC 28
IF(S1 EQ '04') EXEC 28
IF(S1 EQ '05') EXEC 28
IF(S1 EQ '06') EXEC 28
COMMENT
COMMENT ********INVALID LAB NUMBER********
COMMENT
COMMENT PLEASE RE-ENTER THE LAB NUMBER AGAIN.
EXEC 14
SET VALUE S2 SUBSTR(GROUP,3,1)!!USER
COMMENT WHAT IS YOUR LAST NAME?
READ STRING S4
COMMENT WHAT IS THE NAME OF YOUR LAB FILE?
READ STRING S3
IF ( VERIFY(S3,'$') EQ 2 ) COPY FROM %S3 TO 10 BY 10
IF ( VERIFY(S3,'$') EQ 1 ) COPY FROM $%S3 TO 10 BY 10
POINT ' JOB '
IF (CURRENT LT 0) EXEC 40
SET VALUE W1=*
POINT '$JOB''
SET VALUE W2=*
DEL $W1/$W2
POINT '// '
IF (CURRENT GT 0) EXEC 44
SET VALUE W3=*
99999.99 //
0.001 //   JOB
0.002 /*JOBPARM   V=D
0.003 //PROCLIB  DD  DSN=TS0618.PROCLIB
0.004 //       EXEC  INTERNAL
0.005 //PROGRAM  DD  *
0.006 $JOB      %S1%S2     %S4
RUN
COMMENT LAB %S1 HAS BEEN ANALYZED UNDER THE TC%S2 NUMBER.
SET ESCAPE ''
CLEAR ACTIVE
CLEAR EXEC
CLEAR ACTIVE
SET ESCAPE %
COMMENT WHICH OF THE FOLLOWING LAB NUMBERS DO YOU WANT ANALYZED?
COMMENT
COMMENT       L1 L2 L3
COMMENT
READ STRING S1
IF(S1 EQ 'L1') EXEC 28
IF(S1 EQ 'L2') EXEC 28
IF(S1 EQ 'L3') EXEC 28
COMMENT
COMMENT ********INVALID LAB NUMBER********
COMMENT
COMMENT PLEASE RE-ENTER THE LAB NUMBER AGAIN.
EXEC 63
```

CIS 212

Instructions to run the Analyzer

Run your final version of the program to get the printout for submission. AFTER YOU GET THE FINAL AND CORRECT OUTPUT, USE THE FOLLOWING STEPS TO RUN THE ANALYZER.

1. Clear your active file by using the Command: CLEAR ACTIVE

2. Type the following command:

Command? ANALYZE

After you have typed the ANALYZE command and pressed the RETURN key, you will be asked the following questions. Simply insert the answer to these questions.

Questions would be asked	Answers to be submitted
a. What course is this? (212 or 313)	a. Enter 212
b. Which of the following lab numbers do you want analyzed? 02 03 04 05 06	b. If you want to analyze Lab2, Type 02 If you want to analyze Lab3, Type 03 If you want to analyze Lab4, Type 04 If you want to analyze Lab5, Type 05 If you want to analyze Lab6, Type 06
c. What is your Last Name?	c. Enter your Last Name
d. What is the name of your Lab file?	d. Enter the actual file name that you have used to save your Lab. (For example, if you are analyzing Lab2, and the actual file name for Lab2 is PROG2 then enter PROG2).

After you have answered these questions, wait till you get the following message:

LAB (02 or 03 or 04 or 05 or 06) HAS BEEN ANALYZED UNDER THE TRnnnn NUMBER

Where nnnn is the 4-digit number of your personal user-id. As soon as you get this message, you are done with the analyzer.

If you have any question, comment or difficulty, please see your instructor or Mr. DEBNATH in CL 418.

THANK YOU and GOOD LUCK

APPENDIX - C

Maintenance Procedure

The Software Science COBOL Analyzer developed at OSU is a significant component of the Software Metrics Research Group in the Department of Computer and Information Science. The Analyzer is used to collect data from the students of undergraduate COBOL classes (e.g. CIS212, CIS313), needed to pursue further research in Software Science. In addition to the students' programs, various kinds of COBOL programs are also collected for analysis from the University Systems Computer Center as well as from other organizations. Therefore, the person responsible for maintaining the Analyzer has direct interaction with many different groups of people. The present section provides a few of the major steps to be followed for maintenance of the Analyzer. Particular attention is given to its interface with undergraduate COBOL classes.

Procedure:

1. It is important to become familiar with the working of the analyzer program as well as to know how to use the analyzer in the OSU environment.
2. The handout containing the necessary instructions to run a job through the analyzer should be provided to each student at the beginning of the quarter (see sample handout in Appendix-B)
3. CREATE THE DISK SPACE

Before the students start running their jobs through the analyzer, space should be created on the disk in order to collect the students' output from the analyzer. Currently the disk file called `CIS212.FREQ' is used for this purpose. The disk space for the file `CIS212.FREQ' is created according to the SPACE parameter used in the JCL file called WIDJET (see Appendix-B); e.g.

SPACE = (TRK, (50,20)) , DISP = (MOD, CATLG)

Thus, the size of this disk file can be changed simply by changing the SPACE parameter as desired.

4. ARCHIVE THE DATA ON THE DISK

Disk space is very expensive and the collected data on the disk is not usually used during the quarter. Therefore, these data must be archived to tape using the ASM2 commands [IRCC ASC2 Manual] quite frequently. It is strongly suggested that every 50-100 tracks of data should be archived to tape. The archive commands, $AR, provides the user with the ability to archive the specified data set(s) to tape.

Example : $AR `DSLIST'

`DSLIST' is the list of input data set names

5. ASSIGN APPROPRIATE NAME AND MAXIMUM RETENTION PERIOD FOR THE FILE TO BE ARCHIVED

While archiving the data from the disk, the file should be assigned a name which is different from the original disk file name and from any previously archived file name. The new file name should reflect the name of the quarter when the data was collected. This helps recognize the file easily when using it after a long period of time. One should also specify the maximum retention period (e.g. 365 days) for the file being archived to tape.

The archive command [asm2 Manual] to be used in order to satisfy the requirements of step4 and 5 has the following syntax.

$AR `DSLIST´ RETPD (`INTEGER´) QUAL (`QUALIFIER´)

Operands:

`DSLIST´ - List of input data set names.

RETPD (`INTEGER´)

- Specifies the desired retention period on tape

- `INTEGER´ - a 1 to 3 digit value which defines the number of days the user wishes to have the specified data set retained on archive tape.

QUAL (`QUALIFIER´)

- causes the data set to be renamed when archived

`QUALIFIER´ - a 1 to 8 character string in which the first character must not be numeric.

Example:

$AR CIS212.FREQ RETPD (365) QUAL(FALL1981)

This example illustrates that the disk file named CIS212.FREQ would be archived to tape with a new name CIS212. FALL1981.FREQ, and that a 365 day retention period would be in effect.

6. RETRIEVE THE ARCHIVED DATA SET AND PRODUCE QUARTERLY REPORT.

At the end of the quarter, the data collected for the entire quarter should be restored on the disk to produce two different quarterly reports. The first report provides information concerning which students have run which course labs through the analyzer. This report can be provided to the instructors of the course in case course grades are influenced by the students' use of the analyzer. The second report summarizes the Software Science metrics collected for all the programs during the quarter, and is used by the software metrics research group.

Both reports require some sorting of the data. The procedure for producing the reports is described below.

(a) The sort keys for the first report is as follows

Key1 : Students account number

Key2 : Lab number.

It is desirable that each page of this report contain the account number of a particular student and the lab numbers which were run by this student during the whole quarter. This format makes it easier for the course instructors to use this information for grading purposes. Presently, a program called "BONUSREP" generates this report. This report should be distributed to all the course instructors before the final examination begins.

(b) The second report should be sorted according to lab numbers so that all the data for each lab appear together. This makes the analysis convenient. Currently, a program named "REP" is used to produce this quarterly report.

The ASM2 command $RA (Reload from Archives) can be used to restore an archived data from an archived tape to an on-line disk pack, and has the following syntax.

$RA `DSLIST´

Example:

$RA CIS212.FALL1981.FREQ

The use of this reload command will allow the desired data sets to be reloaded into the disk. After transfering all the required data to the disk, the programs "REPBONUS" and "REP" can be run on these data for producing the above reports.

The following wylbur commands are used to run "REP" and "REPBONUS".

Command? SET GROUP TS1

Command? SET USER 483

Command? USE REP (or REPBONUS) ON CATLG

Command? RUN

7. FREE UP THE UNNECESSARY DISK SPACE.

After producing the quarterly reports, all disk spaces must be freed as quickly as possible, either by scratching all the data sets or by archiving these back to tape depending on whether a copy of the data sets already exist on tape or not.

The following Wylbur command is used to scratch a data set from the disk.

Command? SCRATCH (Data set name)

The same $AR command, as discussed above, is used to archive the data set to tape.

Note that maintenance of the analyzer involves handling a large number of important files existing on disk as well as on tape. In order to maintain these files efficiently, one should remember that only those files which are used very often should be kept on disk; other files should be saved on tape. The retention period of all files on tape MUST be checked periodically by using the $AI (Archive Catalog Information) Command [ASM2 Manual], and the expiration date must be updated if necessry. Another possibility is that the data can be saved on a personal tape. Otherwise, it is possible that important data will be lost.

Finally, the programs collected from sources other than OSU courses should be run through the analyzer using the necessary JCL, and the output should be saved on separate files, if possible. The present JCL called "JCLTAPE" is used to run the programs that are collected from the University Systems Computer Center and from other organizations. The output from the analyzer is collcted on the separate disk file called "US.FREQ". Since the source programs for these analyses exist on a

personal tape, it is possible to scratch the disk file containing the results of the analyses after producing the final report (report #2 described earlier in this section). NO archiving of the analysis data is necessary for these programs.

APPENDIX - D

INVOKING THE PURDUE ANALYZER AT IRCC

D.1: JCL REQUIRED TO RUN THE PURDUE ANALYZER

For comparison purposes a COBOL analyzer written at Purdue University can be run at the IRCC of Ohio State University using the following JCL.

```
//JOB
//REGION=512K
/*JOBPARM  LINES=7000,DISKIO=5000
//SA EXEC COBSORT
//COB.SYSIN  DD*
        .
        .
        .
        .
        .
  (TS1483.COBOL.KEYWORD.FILE file)
        .
        .
        .
        .
/*
//GO.SORTFIL  DD UNIT=SYSVIO,SPACE = (CYL,(1,1))
//GO.KEYWD  DD  DSN=TS1483.KEYWORDS.DATA,
                UNIT=USERDA,DISP=(NEW,CATLG),
                SPACE=(CYL,(1,1)),
                DCB=(RECFM=FB,LRECL=30,BLKSIZE=300)
//SECOND EXEC COBUCG, TIME=10
//COB.SYSIN  DD *
        .
        .
        .
        .
        .
        .
(TS1483.PURDUE.COBOL.ANALYZER file)
        .
        .
        .
        .
        .
        .
/*
```

```
//GO.SYSIN  DD
        .
        .
        .
        .
        .
        .
(COBOL program)
        .
        .
        .
/*
//GO.KEYWD  DD DSN=TS1483.KEYWORDS.DATA,UNIT=USERDA, DISP=SHR
//GO.OUTPUT  DD  SYSOUT=A
//          DCB=(RECFM=FBA,LRECL=133,BLKSIZE=133)
//GO.SUMFILE  DD SYSOUT=A,
//          DCB=(RECFM=FBA,LRECL=133,BLKSIZE=133)
```

The Purdue Analyzer consists of two separate programs, namely, SORT-TAB and ANALYZE. Both of these files have been compiled with necessary modifications using the standard COBOL compiler at Ohio State University, and renamed as TS1483.COBOL.KEYWORD.FILE and TS1483.PURDUE.COBOL.ANALYZER, respectively. The first program generates a file of predefined COBOL keywords and noisewords, while the second program analyzes any COBOL source program given as input using the counting strategy defined by the first program.

One should note that the COBOL program to be analyzed should include both the Data and Procedure divisions. Separate analyses will be done for the two divisions, using the predefined counting strategy developed at Purdue University.

D.2: OUTPUT OF THE PURDUE ANALYZER

The output of the Purdue Analyzer consists of the listing of the input file followed by the statistics for the entire program. Statistics include the list of operators and operands along with their corresponding frequency counts for both the Data and Procedure divisions. The summary of the basic software metrics values, namely, ETA1, N1, ETA2, N2, N, NHAT and REL ERROR are also produced. A sample output generated by the Purdue analyzer is included for completeness.

STATISTICS FOR THIS MODULE

DATA DIVISION

OPERATOR	FREQUENCY
.	97
ASSIGN	4
BLOCK	1
FD	4
.	.
.	.
.	.
.	.
	ETA1=____
	N1= ____

OPERAND	FREQUENCY
PRINTER-FILE1	2
PRINTER-FILE2	2
X(8)	4
CURR-DATE	2
0	13
PR-MM	1
9	10
.	.
	ETA2= ___ _
	N2 = _____

PROCEDURE DIVISION

OPERATOR	FREQUENCY
ACCEPT	1
ADD	16
CLOSE	1
MOVE	76
OPEN	1
.	.
.	.
.	.
.	.
.	.
.	.
.	.
PERFORM EDIT-CHECK	1
PERFORM HEADER-PR1	1

ETA1 =
N1 =

OPERAND	FREQUENCY
PRINTER-FILE1	2
PRINTER-FILE2	2
31	1
'*'	1
CURRENT-DATE	1
CURR-DATE	2
.	.
.	.
.	.
.	.
.	.

ETA2 =
N2 =

ETA2 = _____
N2 = _____

	DATA	PROCEDURE	MODULE
ETA1	8	31	39
N1	238	269	507
ETA2	74	84	94
N2	159	358	517
N	397	627	1024
NHAT	483	690	822
REL ERROR(N)	0.1780	0.0913	0.2457

D.3: COMPARISON BETWEEN OSU AND PURDUE ANALYZER

The following differences have been observed between the outputs of the two analyzers.

	PURDUE	OSU
DD Operator:	ASSIGN and SELECT are treated as DD operators. LABEL, RECORD and STANDARD have been skipped.	ASSIGN and SELECT are not taken into consideration. Instead LABEL, STANDARD are treated as DD operators.
DD Operands:	FILLER and labels are not counted; Some x(n)'s in PIC's are not counted also.	FILLER, labels and all x(n)s are counted.
PD Operators:	Operators such as TO, INTO, NEXT, AFTER FROM, INPUT, OUTPUT, CORRESPONDING, EOS are skipped altogether from the operator operand list.	all the standard operators are counted; major differences come from EOS is TO counts. EOS is the number of verbs.
	NUMERIC is treated as an operator.	NUMERIC is treated as an operand.
	EQUAL, NOT EQUAL are treated as separate operators. Similarly GREATER, NOT GREATER etc.	NOT and EQUAL are separated and treated as two different operators.
	Instead of End of statements (EOS), Purdue counts required periods.	

PD Operand counting is very comparable, except that in the OSU analyzer there are some additional operands e.g. SENTENCE, OF.

It is noted that (1) the main difference in counts come from DD operands and PD operators. (2) The Purdue analyzer does not count PD operators like TO, FROM (which are used with arithmetic verbs and sometimes are not optional e.g. in Move A TO B, TO is not optional but in A EQUAL TO B, TO is optional; there are many such examples). OSU counts all these required operators.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER
17150.3-EL

2 GOVT ACCESSION NO.
AD-A130599

3 RECIPIENT'S CATALOG NUMBER

4 TITLE (and Subtitle)
A Software Science Analyzer for COBOL

5 TYPE OF REPORT & PERIOD COVERED
Technical

6 PERFORMING ORG. REPORT NUMBER

7 AUTHOR(s)
K. C. Fung
N. C. Debnath
S. H. Zweben

8 CONTRACT OR GRANT NUMBER(s)
DAAG29 80 K 0061

9 PERFORMING ORGANIZATION NAME AND ADDRESS
Ohio State University
Columbus, OH 43210

10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11 CONTROLLING OFFICE NAME AND ADDRESS
U. S. Army Research Office
Post Office Box 12211
Research Triangle Park, NC 27709

12 REPORT DATE
1982

13 NUMBER OF PAGES
118

14 MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)

15 SECURITY CLASS. (of this report)
Unclassified

15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES
The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)
An analyzer of COBOL programs which computes the metrics from software science is deseribed. The report discusses the overall design of the analyzer, including detailed descriptions of each of its modules. It also contains instructions for the use and maintenance of the analyzer at Ohio State University.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

END

DATE
FILMED

8-83

DTIC